O'REILLY®

TURING 图灵程序设计丛书



React Native 开发指南

Learning React Native

从基础开始逐步深入，利用React Native成功部署可100%代码复用的跨平台应用。

[美] Bonnie Eisenman 著

黄为伟 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

黄为伟

东北大学软件工程国际班大三学生，目前是BitTitan公司的软件开发实习生，曾在阿里巴巴FGT前端团队实习。开源项目react-native-guide作者，热爱Web技术。

TURING 图灵程序设计丛书

React Native开发指南

Learning React Native
Building Native Mobile Apps with JavaScript

[美] Bonnie Eisenman 著
黄为伟 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（CIP）数据

React Native开发指南 / (美) 艾森曼
(Bonnie Eisenman) 著 ; 黄为伟译. -- 北京 : 人民邮
电出版社, 2016.6
（图灵程序设计丛书）
ISBN 978-7-115-42526-3

Ⅰ. ①R… Ⅱ. ①艾… ②黄… Ⅲ. ①移动终端－应用
程序－程序设计－指南 Ⅳ. ①TN929.53-62

中国版本图书馆CIP数据核字(2016)第118178号

内 容 提 要

本书通过丰富的示例和详细的讲解，介绍了React Native这款JavaScript框架。在React Native中利用现有的JavaScript和React知识，就可以开发和部署功能完备的、真正原生的移动应用，并同时支持iOS与Android平台。除了框架本身的概念讲解之外，本书还讨论了如何使用第三方库，以及如何编写自己的Java或Objective-C的React Native扩展。

本书适合前端工程师或Web开发者，以及希望开发跨平台移动应用的其他开发人员。

◆ 著　[美] Bonnie Eisenman
译　黄为伟
责任编辑　朱　巍
执行编辑　温　雪
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：14.75
字数：349千字　2016年6月第1版
印数：1－3 500册　2016年6月北京第1次印刷

著作权合同登记号　图字：01-2016-3664号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第8052号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

本书将介绍 React Native，一款由 Facebook 公司出品的用来构建移动应用的 JavaScript 框架。在 React Native 中利用现有的 JavaScript 和 React 知识，就可以开发和部署功能齐全的、真正原生的移动应用，并同时支持 iOS 与 Android 平台。采用 JavaScript 作为开发语言并不意味着需要退而求其次，相反，React Native 在不牺牲原生样式和体验的前提下，相比传统移动开发仍然有很多优势。

我们将从基础开始学习，然后逐步深入，最终部署一款 100% 代码复用的成熟的移动应用到 iOS 应用商店和 Google Play 商店。除了框架本身的概念讲解之外，我们还将讨论如何使用第三方库，以及如何编写自己的 Java 或 Objective-C 的 React Native 扩展。

如果你想从前端工程师或 Web 开发者的视角接触移动应用开发，那么本书就是为你量身定做的。React Native 是一款令人惊奇的框架，愿你怀着和我一样喜悦的心情来探索它。

预备知识

本书总体上不是介绍 React 的，我们假设你对 React 已经有一些了解。如果你从未接触过 React，我建议你在正式开始学习移动开发之前先阅读一两篇相关的教程，尤其应该熟悉 React 的属性（`props`）和状态（`state`）、组件的生命周期以及如何创建 React 组件等知识。

同时，我们也会使用一些 ES6 和 JSX 的语法。如果你对这些还不太熟悉也没有关系，我们将在第 2 章讲解 JSX，在附录 A 中介绍 ES6 的语法。这些语法本质上与你习惯的 JavaScript 代码是一对一的解析关系。

本书假设你使用 OS X 操作系统进行开发。开发 iOS 应用必须使用 OS X 操作系统。使用 Linux 和 Windows 开发 Android 应用的支持工作仍在进行中，你可以在官网（http://facebook.github.io/react-native/docs/linux-windows-support.html）阅读更多关于 Linux 与 Windows 支持的相关内容。

排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语。

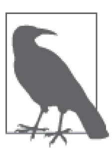
- 等宽字体（`Constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`Constant width bold`**）
  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`Constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/bonniee/learning-react-native 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产

品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Learning React Native* by Bonnie Eisenman (O'Reilly). Copyright 2016 Bonnie Eisenman, 978-1-491-92900-1"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例

代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920041511.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

资源

单枪匹马会让学习过程变得困难；虽然事实并不一定如此，但你不一定要这样做。这里有一些资源，也许在阅读过程中会给你带来一些帮助。

- 本书中所有的代码示例都在 GitHub 代码仓库中（https://github.com/bonniee/learning-react-native），如果在学习过程中遇到困难或者需要代码的上下文，不妨看看这里。
- 加入 LearningReactNative.com（http://learningreacnative.com）的邮件列表获取后续的文章、建议和实用的资源。
- 官方文档（https://facebook.github.io/react-native/）中有大量优秀的参考资料。

此外，React Native 社区也是实用的资源：

- Brent Vatne 的 React Native newsletter（http://brentvatne.ca/react-native-newsletter/）
- Stack Overflow 上的 react-native 标签分类（http://stackoverflow.com/questions/tagged/react-native）
- Freenode 上的 #reactnative 小组（irc://chat.freenode.net/reactnative）

致谢

写成本书离不开大家的帮助和支持。首先要感谢我的编辑 Meg Foley 以及 O'Reilly 团队中的其他成员。同时也要感谢技术审稿人 David Bieber、Jason Brown、Erica Portnoy 和 Jonathan Stark，他们花费了大量时间审阅本书并提出了深有见地的反馈意见。感谢 React Native 团队，如果没有他们杰出的产品，自然也不会有本书的诞生。感谢 Zachary Elliot 对 Zebreto 应用和 Android 开发提供的帮助。

最后，非常感谢我亲爱的朋友们，在本书写作过程中给了我无限的包容、莫大的精神支持和悉心的指导，并在需要的时候陪我消遣。谢谢你们！

第 1 章

初识React Native

React Native 是一款用来开发真正原生、可渲染 iOS 和 Android 移动应用的 JavaScript 框架。它基于 Facebook 公司开源的 JavaScript 用户界面开发框架 React 而产生，但 React 将浏览器作为渲染平台，而 React Native 的渲染平台则是移动设备。也就是说，Web 开发者现在就可以使用我们非常熟悉的 JavaScript 类库来开发真正原生的移动应用。并且，由于编写的大部分代码可以在平台之间共享，React Native 可以让你更简单地同步开发 Android 和 iOS 应用。

与 Web 平台上的 React 相似，React Native 也使用 JSX 进行开发，这种编程语言结合了 JavaScript 和类 XML 标记语言。React Native 在后台通过“桥接”的方式调用由 Objective-C（iOS 平台）或 Java（Android 平台）开放的原生渲染接口，因此，你的应用将会使用真正原生的移动 UI 组件，而不是传统的 WebView 渲染方式，进而拥有与其他移动应用一样的外观和体验。同时，React Native 也为 JavaScript 开放了平台接口，让你的应用能够使用平台提供的功能，例如摄像头和用户定位等。

React Native 目前同时支持 iOS 和 Android，今后也可能扩展到其他平台上。在本书中，我们将会同时介绍 iOS 和 Android 的知识，并且书中大部分代码都能跨平台运行。没错，你完全可以用 React Native 来开发用于正式发布的移动应用。据了解，Facebook（https://code.facebook.com/posts/1014532261909640/react-native-bringing-modern-web-techniques-to-mobile/）、Palantir（https://medium.com/@clayallsopp/react-native-in-production-2b3c6e6078ad#.wui5g18dx）和 TaskRabbit（http://tech.taskrabbit.com/blog/2015/09/21/react-native-example-app/）等公司已在使用它开发面向用户的应用。

1.1 React Native的优点

事实上，React Native 调用宿主平台标准渲染接口的方式已经使它从其他现有的跨平台应用开发方案（比如 Cordova 或 Ionic）中脱颖而出。目前通过编写 JavaScript、HTML 和 CSS 的方式进行应用开发的方案大多使用 WebView 进行界面渲染，当然这种方案是可行的，但也带来了一些问题，尤其是性能损耗。同时，这种方案通常无法使用宿主平台的原生 UI 组件，所以这些框架尝试去模仿原生 UI 组件的行为，而模仿的效果通常让人觉得不够真实。为了模仿各种类似动画这样的细节，一般都要付出巨大的努力，然而它们很快又会过时。

相反，React Native 则将你的代码解析成真正原生的 UI 组件，利用了所用平台上现有的视图渲染方式。并且，由于 React 不在 UI 主线程中运行，你的应用可以在不牺牲灵活性的前提下保持高性能。React Native 的生命周期与 React 相同，当属性（`props`）或状态（`state`）发生改变时，React Native 会重新渲染视图。而与浏览器上的 React 最大的不同在于，React Native 使用了宿主平台上的 UI 元素来代替 HTML 和 CSS。

对于习惯了 Web 平台的 React 开发者来说，这意味着你可以使用熟悉的工具来开发真正原生的移动应用。在开发者体验与跨平台开发等方面，React Native 较传统的移动端开发来说也有一定的优势。

1.1.1 开发者体验

如果你曾经有过移动端的开发经历，将会对 React Native 的易用性感到震惊。React Native 团队已经研发了强大的开发工具并在框架内嵌入了友好的错误提示，因此使用这些强大的工具会让开发体验更加自然。

例如，由于 React Native 使用了 JavaScript，我们查看修改结果时不需要重新编译。相反，按下 Command+R 就可以刷新应用，就和在网页上开发一样。在传统移动端开发中，编译构建应用所花费的时间会积少成多，相比之下 React Native 的快速迭代就像是天赐之福。

React Native 还可以让你更好地利用智能调试工具以及错误报告机制。如果你习惯于使用 Chrome 或者 Safari 的开发工具（图 1-1），那么使用它们进行移动开发一定也会让你十分愉悦。同样，你可以选择喜爱的任何文本编辑器来开发 JavaScript：React Native 不强制你使用 Xcode 进行 iOS 开发，也不强制使用 Android Studio 进行 Android 开发。

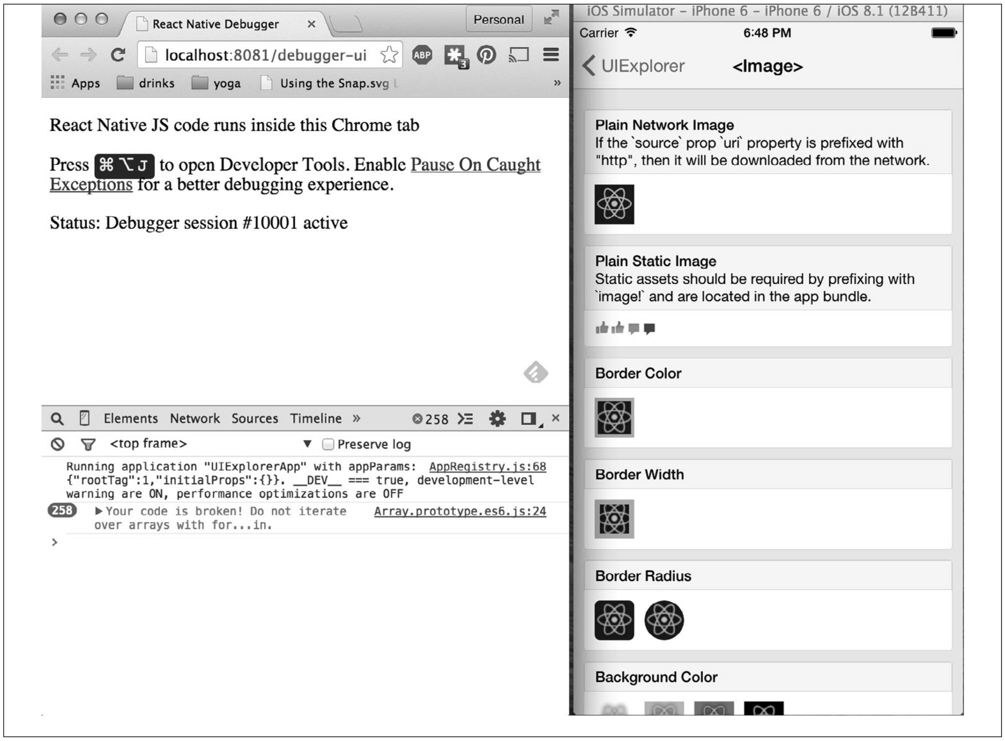


图 1-1：使用 Chrome 调试器

除了能逐渐改善开发者体验之外，React Native 也极有可能给你的产品发布周期带来一些积极的影响。例如，Apple 公司允许通过网络对基于 JavaScript 开发的功能进行更新，无需额外的审核周期。

所有这些小福利将会节省你和你的伙伴们的时间和精力，让你可以专注于工作中那些更有趣的部分，同时也能提高你的工作效率。

1.1.2 代码复用与知识共享

使用 React Native 可以大大减少开发移动应用所需的资源。任何了解如何编写 React 的开发者现在都可以使用相同的技能同时开发 Web 应用、iOS 应用和 Android 应用。React Native 避免了按平台分工的必要，可以让你的团队更加快速地迭代产品，并更加高效地共享知识和资源。

除了知识的共享之外，你的大部分代码也可以被共享。当然，不是你写的所有代码都可以做到跨平台；这取决于你需要在特定的平台上实现什么功能，你可能偶尔也需要涉及 Objective-C 或 Java 的知识（好在这也不是很糟糕，我们将会在第 7 章讲解本地模块的用法）。使用 React Native，在不同平台之间复用代码将会变得出乎意料地简单。例如，React

Europe 2015 大会（https://www.youtube.com/watch?v=PAA9O4E1IM4&feature=youtu.be）提到，Facebook Ads Manager 这款 Android 应用共享了其 iOS 版本 87% 的代码。另外，我们通过本书完成的一款闪卡应用做到了 iOS 和 Android 代码的完全复用。这是很难超越的成就！

1.2 风险和缺点

就像世间万物一样，React Native 也难免存在一些缺点，至于 React Native 是否适合你的团队，则取决于你们自身的情况。

React Native 于 2015 年 3 月发布了对 iOS 平台的支持，同年 9 月开始支持 Android 平台。由于目前 React Native 项目还很年轻，不够成熟可能是其最大的风险。它的文档确实还有提升的空间，同时项目也在不断升级和改进。一些特性在 iOS 和 Android 平台上仍未得到支持，社区也在不断寻找最佳的开发实践。不过，好在大多数情况下你都可以自己实现那些缺少的接口，我们也会在第 7 章讨论相关内容。

React Native 在你的项目中引入了新的一层，因此带来了一些调试上的麻烦，尤其是在 React 和宿主平台交互时。我们将在第 8 章更加深入地讲解 React Native 的调试技巧，并探讨一些常见的问题。

React Native 依然还很年轻，追随新技术时可能遇到的问题在此也不可避免。不过总体来说，我觉得你将会看到它带来的收益大于风险。

1.3 小结

React Native 是一个振奋人心的框架，它使得 Web 开发者可以使用他们现有的 JavaScript 知识开发出强大的移动应用。在不牺牲用户体验和应用质量的前提下，React Native 提高了开发效率，提供了在 iOS、Android 和 Web 平台上的代码共享。由于它依然很新并且还在持续不断地更新，你在使用时需要作一番权衡。如果你的团队可以解决新技术带来的不确定问题，并且想开发跨平台的应用，那么不妨试试 React Native 吧。

在下一章中，我们将看一看 React Native 与 React 主要有哪些不同，并讲解一些关键的概念。如果你想跳过这个部分，可以直接跳到第 3 章的实战部分，第 3 章会从开发环境的搭建讲起，着手开发我们的第一个 React Native 应用。

第 2 章

React Native工作原理

在本章中，我们将介绍“桥接”的知识，了解 React Native 在后台是如何工作的，再看看 React Native 组件与 Web 平台上的组件有何区别，以及开发移动应用所需的组件创建与样式的知识。

> 如果想学习 React Native 的实战部分，你可以直接跳到下一章。

2.1 React Native是如何工作的

使用 JavaScript 开发移动应用的想法可能有些奇怪。在移动环境中使用 React 是怎样实现的呢？为了更好地理解 React Native 的工作原理，我们首先需要回顾一下 React 的一个特点：Virtual DOM（虚拟 DOM）。

在 React 中，Virtual DOM 就像是一个中间层，介于开发者描述的视图与实际在页面上渲染的视图之间。为了在浏览器上渲染出可交互的用户界面，开发者必须操作浏览器的 DOM（Document Object Model，文档对象模型）。这个操作代价昂贵，对 DOM 的过度操作将会给性能带来严重的影响。React 维护了一个内存版本的 DOM，通过计算得出必要的最小操作并重新渲染。图 2-1 展示了这个工作过程。

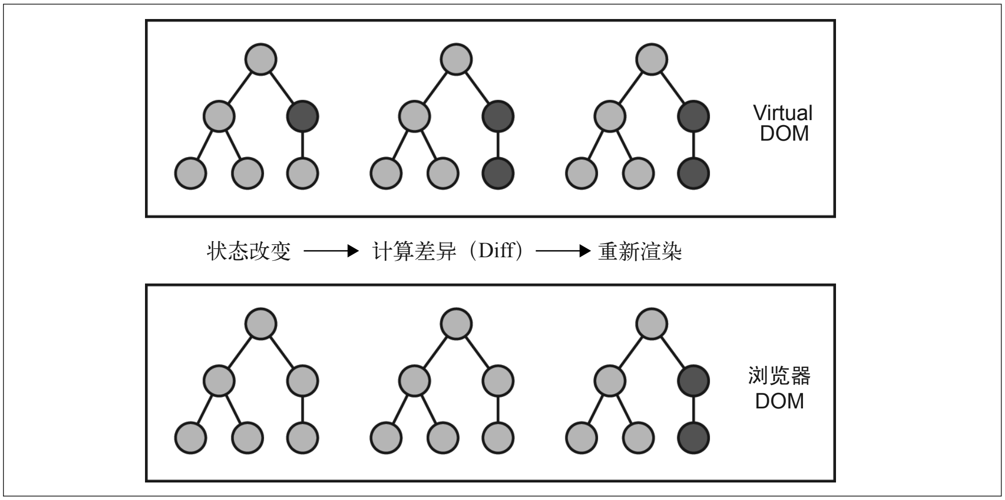


图 2-1：执行 Virtual DOM 的计算，减少浏览器 DOM 的重复渲染

对于 Web 环境的 React 而言，大多数的开发者认为 Virtual DOM 的出现主要是为了优化性能。Virtual DOM 确实能提升性能，但它主要的潜力在于提供了强大的抽象能力。在开发者的代码与实际的渲染之间加入一个抽象层，这带来了很多可能性。想象一下，如果 React 能够渲染到浏览器以外的其他平台呢？毕竟，React 已经“理解”了你的应用应该如何展现。

确实，这就是 React Native 的工作原理，如图 2-2 所示。React Native 调用 Objective-C 的 API 去渲染 iOS 组件，调用 Java 接口去渲染 Android 组件，而不是渲染到浏览器 DOM 上。这使得 React Native 不同于那些基于 Web 视图的跨平台应用开发方案。

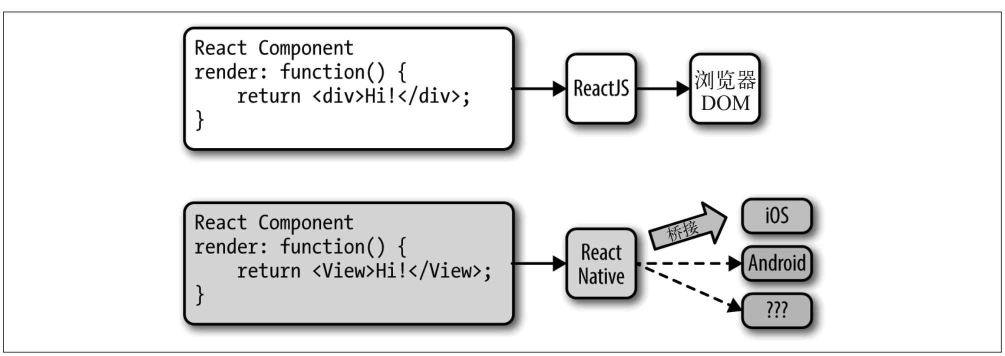


图 2-2：React 可以渲染到多平台

“桥接”令这一切成为可能，它使得 React 可调用宿主平台开放的 UI 组件。React 组件通过 `render` 方法返回了描述界面的标记代码。如果是在 Web 平台上，React 最终将把标记代码解析成浏览器的 DOM；而在 React Native 中，标记代码会被解析成特定平台的组件，例如

<View> 将会表现为 iOS 平台上的 UIView。

React Native 目前同时支持了 iOS 和 Android 两种平台。由于 Virtual DOM 提供了抽象层，React Native 也可以支持其他平台，只需为其提供“桥接”即可。

2.2 渲染周期

如果你习惯使用 React，那你应该熟悉 React 的生命周期。当 React 在 Web 环境中运行时，渲染周期始于 React 组件挂载之后（图 2-3）。

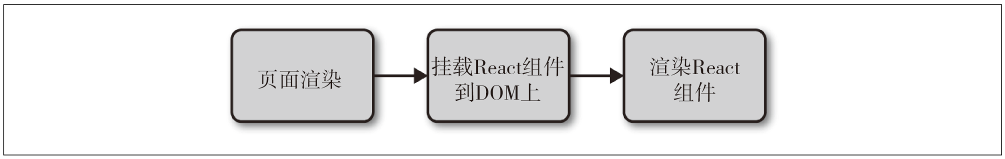


图 2-3：React 组件挂载过程

接着，React 进入渲染周期并根据需要渲染组件（图 2-4）。

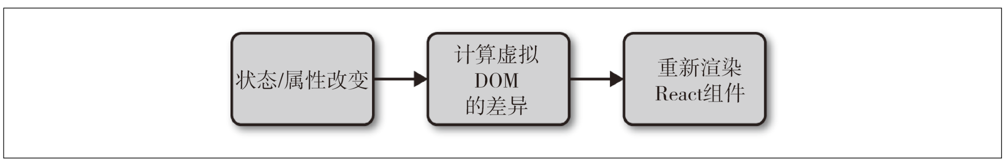


图 2-4：React 组件重新渲染过程

在渲染阶段，React 将开发者在 render 方法中返回的 HTML 标记直接按需渲染到页面上。

至于 React Native，生命周期与 React 基本相同，但渲染过程有一些区别，因为 React Native 依赖于“桥接”，正如先前图 2-2 所示。JavaScript 通过“桥接”的解析，间接调用宿主平台的基础接口和 UI 元素（也就是 Objective-C 或 Java）。由于 React Native 不是在 UI 主线程运行，它可以在不影响用户体验的前提下执行这些异步调用。

2.3 在React Native中创建组件

所有的 React 代码都存在于 React 组件中。React Native 组件与 React 组件大体上一致，但在渲染和样式方面有一些重要的区别。

2.3.1 编写视图

当编写 Web 环境的 React 时，视图最终需要渲染成普通的 HTML 元素（<div>、<p>、<span>、<a> 等）。而在 React Native 中，所有的元素都将被平台特定的 React 组件所替换

（见表 2-1）。最基础的组件是能跨平台的 `<View>`，这是一个简单且灵活的 UI 元素，类似于 `<div>` 标签。例如，在 iOS 中，`<View>` 组件被渲染成 `UIView`，而在 Android 平台则被渲染成 `View`。

表2-1：Web与React Native基础元素的比较

React	React Native
`<div>`	`<View>`
`<span>`	`<Text>`
`<li>`, `<ul>`	`<ListView>`
`<img>`	`<Image>`

其他组件则是平台特定的。例如，`<DatePickerIOS>` 组件显然将被渲染成 iOS 标准的日期选择器。下面是从 UIExplorer 示例应用中摘录出来的代码，用来展示 iOS 日期选择器。正如你期待的那样，用法相当直观：

```
<DatePickerIOS
  date={this.state.date}
  mode="date"
  timeZoneOffsetInMinutes={this.state.timeZoneOffsetInHours * 60}
/>
```

以上代码将被渲染成一个标准的 iOS 日期选择器（如图 2-5 所示）。



图 2-5：DatePickerIOS，顾名思义，是 iOS 特有的组件

由于我们所有的 UI 元素均为 React 组件，而不是像 `<div>` 这样基础的 HTML 元素，因此我们在使用每一个组件之前，都需要明确地进行导入。例如，我们可以这样导入 `<DatePickerIOS>` 组件：

```
var React = require('react-native');
var {
```

```
  DatePickerIOS
} = React;
```

UIExplorer 应用是一个打包的标准 React Native 示例（https://github.com/facebook/react-native#examples），可以让你查看它所支持的所有 UI 元素，建议你体验一下其中包含的各种元素。除此之外，它还讲解了许多关于样式和交互的知识。

> 平台特定的元素和接口在官方文档中有特殊的标签，通常使用平台名称作为后缀，例如：`<SwitchAndroid>` 和 `<SwitchIOS>`。

这些组件因平台而不同，因此在使用 React Native 时，如何组织你的组件变得尤为重要。在 Web 环境的 React 中，我们通常混合各种 React 组件，有的组件控制逻辑及其子组件，而有的则渲染原生标记。在使用 React Native 时，如果你想复用代码，那么这些组件的抽象分离就至关重要。当然，如果一个组件渲染 `<DatePickerIOS>` 元素，那它显然不能在 Android 平台复用了。不过，如果一个组件封装的是关联逻辑，那就可以被复用。因此，视图组件可以根据平台进行替换选择。如果你乐意的话，还可以为组件设计平台特定的版本，例如 picker.ios.js 和 picker.android.js。我们将在 4.4.2 节具体讲解。

2.3.2 使用JSX

与 React 中相一致，React Native 也是通过编写 JSX 来设计视图，并将视图标记和控制逻辑组合在一起成为一个文件。React 刚问世的时候，JSX 在业界引起了强烈的反响。对于许多 Web 开发者来说，根据技术进行文件分离是理所当然的：保持 CSS、HTML 和 JavaScript 文件的独立。然而将标记、控制逻辑，甚至样式合并成一门语言难免会让人觉得混乱。

JSX 认为减少心智负担比文件分离更有用。在 React Native 中，这一点表现得更为明显。在一个没有浏览器的世界里，每个组件的样式、标记和行为被统一成单个文件的形式将会更有意义。因此，React Native 中的 .js 文件实际上就是 JSX 文件。如果你正在使用原生 JavaScript 编写 Web 环境的 React，你应该想转换到 JSX 语法来编写 React Native 项目。

假如你之前从未使用过 JSX，也不用太担心，它非常简单。举个例子，用纯 JavaScript 编写 React 组件的代码看起来如下：

```
var HelloMessage = React.createClass({
  displayName: "HelloMessage",

  render: function render() {
    return React.createElement(
      "div",
      null,
```

```
      "Hello ",
      this.props.name
    );
  }
});

React.render(React.createElement(HelloMessage, { name: "Bonnie" }), mountNode);
```

我们可以通过使用 JSX 使其更为简洁，使用类 XML 标记来代替调用 React.createElement 方法并传入一组 HTML 属性的做法。

```
var HelloMessage = React.createClass({
  render: function() {
    // 返回标记,而不是调用createElement方法。
    return <div>Hello {this.props.name}</div>;
  }
});

// 我们不再需要调用createElement方法。
React.render(<HelloMessage name="Bonnie" />, mountNode);
```

以上两段代码最终都会在页面上被渲染为下面的 HTML：

```
<div>Hello Bonnie</div>
```

2.3.3　原生组件的样式

在 Web 中，正如使用 HTML 标签一样，我们仍然使用 CSS 来为 React 组件添加样式。不论你是否喜欢 CSS，它都已经成为 Web 开发不可或缺的一部分。React 通常不影响我们编写 CSS 的方式，它确实让内联样式（清晰且实用）的使用和样式的动态创建（通过 props 和 state）更加容易。除此之外，React 基本上不关心我们是如何处理样式的。

非 Web 平台上有大量的方法来处理布局和样式。但庆幸的是，我们在使用 React Native 时，只需要用一种标准的方法来处理样式。React 和宿主平台之间的“桥接”包含了一个缩减版 CSS 子集的实现。这个 CSS 子集主要通过 flexbox 进行布局，做到了尽量简单化，而不是去实现所有的 CSS 规则。有别于 Web 平台，CSS 的支持程度因浏览器而不同，React Native 则做到了样式规则的一致。在 React Native 配套的 UIExplorer 应用（https://github.com/facebook/react-native/tree/master/Examples/UIExplorer）中不仅可以查看许多 UI 元素，也能看到许多支持的样式例子。

React Native 也坚持使用内联样式，通过 JavaScript 对象进行样式组织。React 团队先前也提倡在 Web 环境的 React 中使用内联样式。如果你曾经在 React 中使用过内联样式，那么下面的语法你一定非常熟悉了：

```
// 定义一个样式。
var style = {
```

```
    backgroundColor: 'white',
    fontSize: '16px'
  };

  // 然后使用它。
  var tv = (
    <Text style={style}>
      A styled Text
    </Text>);
```

为了让样式更容易管理，React Native 为我们提供了创建和扩展样式的工具。我们将在第 5 章探索这部分内容。

内联样式的写法让你觉得难受？它基于 Web 背景而产生，被公认为标准实践的一个突破。相对于样式表来说，使用样式对象可能需要一些思维上的调整，从而改变你编写样式的方法。然而，在 React Native 中，这是一个实用的转变。我们将在第 5 章讨论编写样式的最佳实践和工作流程，你很难不对它的实际使用效果感到惊讶！

2.4 宿主平台接口

使用 Web 环境的 React 与 React Native 最大的不同应该就在于宿主平台的接口了。在 Web 中，我们遇到的问题通常是由于采纳标准的不一致和碎片化引起的，并且大多数浏览器只支持部分核心的特性。然而在 React Native 中，平台特定的接口在提供优秀原生的用户体验方面发挥了巨大的作用。当然，要考虑的方面还有很多。接口囊括了许多功能，从数据存储到地理服务，以及操控硬件设备（如摄像头）等。由于 React Native 可以扩展到其他平台，我们有机会看到各种不同的平台接口，例如，React Native 和虚拟现实头盔之间的接口会是什么样的呢？

默认情况下，iOS 和 Android 版本的 React Native 支持许多常用的特性，甚至可以支持任何异步的本地接口，本书中将会涉及这些内容。React Native 让宿主平台接口的使用变得更加简单和直观，你可以在其中自由地试验。同时，务必思考一下怎样做才符合目标平台的体验，并在心里设计好交互过程。

毋庸置疑，React Native 的“桥接”不可能暴露宿主平台全部的接口。如果你需要使用一个未支持的特性，完全可以自己动手添加到 React Native 中。另外，如果其他人已经集成，那就更好了，所以应该及时查看社区是否已经或即将支持某些特性。我们将在第 7 章讨论这部分知识。

值得注意的是，使用平台接口也会对代码复用有帮助。同时，实现平台特定功能的 React 组件也是平台特定的。隔离和封装这些组件将会给你的应用带来更大的灵活性。当然，这对你开发 Web 应用同样奏效，如果你想共享 React 和 React Native 的代码，请记住像 DOM 这样的接口在 React Native 中并不存在。

2.5 小结

使用 React Native 为移动应用编写组件与 Web 环境的 React 相比有一些不同。JSX 是强制使用的，并且我们通过创建组件的方式来开发基本模块，比如 `<View>` 代替了 `<div>` 这个 HTML 元素。样式方面也不太一样，我们通过使用 CSS 的子集编写内联语句的方式来编写样式。当然，这些调整都能得到很好的把控。在下一章中，我们将动手编写第一个移动应用！

第 3 章

构建你的第一个应用

本章将会讲解如何搭建 React Native 开发环境以及如何构建一个简单的应用，并将其部署到自己的 iOS 或 Android 移动设备上。

3.1 搭建环境

搭建开发环境让你可以跟着本书的例子一起学习并开发你自己的应用。

关于安装 React Native 的说明可以查看 React Native 官方文档（http://facebook.github.io/react-native/）。官方网站会提供最新的安装参考，不过在此我们也会讲解这些步骤。

你将会用到 Homebrew（http://brew.sh/），一个 OS X 系统的通用包管理工具，用来安装 React Native 的相关依赖。本书假设你使用 OS X 操作系统，因此可以同时开发 iOS 和 Android 应用。

安装好 Homebrew 之后，运行以下命令：

```
brew install node
brew install watchman
brew install flow
```

React Native 包管理器同时使用了 `node` 和 `watchman`，如果在今后的开发过程中遇到问题，建议你更新这些依赖。`flow` 是 Facebook 公司出品的一个类型检查库，它同样被 React Native 所采用（如果你想让 React Native 项目支持类型检查，可以使用 `flow`）。

如果安装过程中遇到问题，你可能需要更新 `brew` 和相关依赖包（以下命令可能比较耗时）。

```
brew update
brew upgrade
```

如果出现错误，你需要修复本地的 `brew` 安装程序，`brew doctor` 可以帮助你找到问题所在。

3.1.1　安装React Native

现在你已经安装好了 `node`，然后就可以通过 npm（Node 包管理器）来安装 React Native 命令行工具了：

```
npm install -g react-native-cli
```

这个步骤将会在你的系统全局安装 React Native 命令行工具。完成之后，祝贺你，此时 React Native 已经安装成功了！

接下来，你需要处理特定平台的安装。为了开发特定平台的移动应用，你需要安装平台开发的依赖。本章将继续讲解相关内容，包括 iOS 和 Android 两个版本。

3.1.2　iOS依赖

为了开发和发布 iOS 应用，你需要获得一个 iOS 开发者账号。申请这个账号是免费的，足够用来开发使用了。如果需要部署到 iOS 应用商店，你需要获得一个许可，价格是每年 99 美元。

如果你还没有完成这一步的话，需要下载并安装 Xcode，它包含了 Xcode 集成开发环境、iOS 模拟器以及 iOS SDK（软件开发工具包）。你可以从应用商店或 Xcode 网站（https://developer.apple.com/xcode/download/）下载。

Xcode 成功安装之后，接受许可，一切就准备就绪了。

3.1.3　Android依赖

Android 依赖的安装需要较多的步骤，应查看官方文档（https://facebook.github.io/react-native/docs/android-setup.html）中最新的安装说明。需要注意的是，这些安装说明都假设你没有安装过 Android 开发环境。总体而言，安装分为三个主要阶段：安装 SDK、安装模拟器工具、创建模拟器。

首先，你需要安装 JDK（Java 开发工具包）和 Android SDK。

(1) 安装最新版本的 JDK（http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html）。
(2) 通过 **`brew install android-sdk`** 安装 Android SDK。
(3) 在 shell 配置文件中正确导出 `ANDROID_HOME` 环境变量（~/.bashrc、~/.zshrc 或其他 shell）。

```
export ANDROID_HOME=/usr/local/opt/android-sdk
```

许多 Android 相关的开发任务都使用这个环境变量。需要确保添加环境变量之后执行 `source` 命令使得配置可以立即生效。

接下来，在命令行执行 **`android`** 命令，从而打开 Android SDK 管理器。如图 3-1 所示，管理器将会显示出开发包的安装情况。

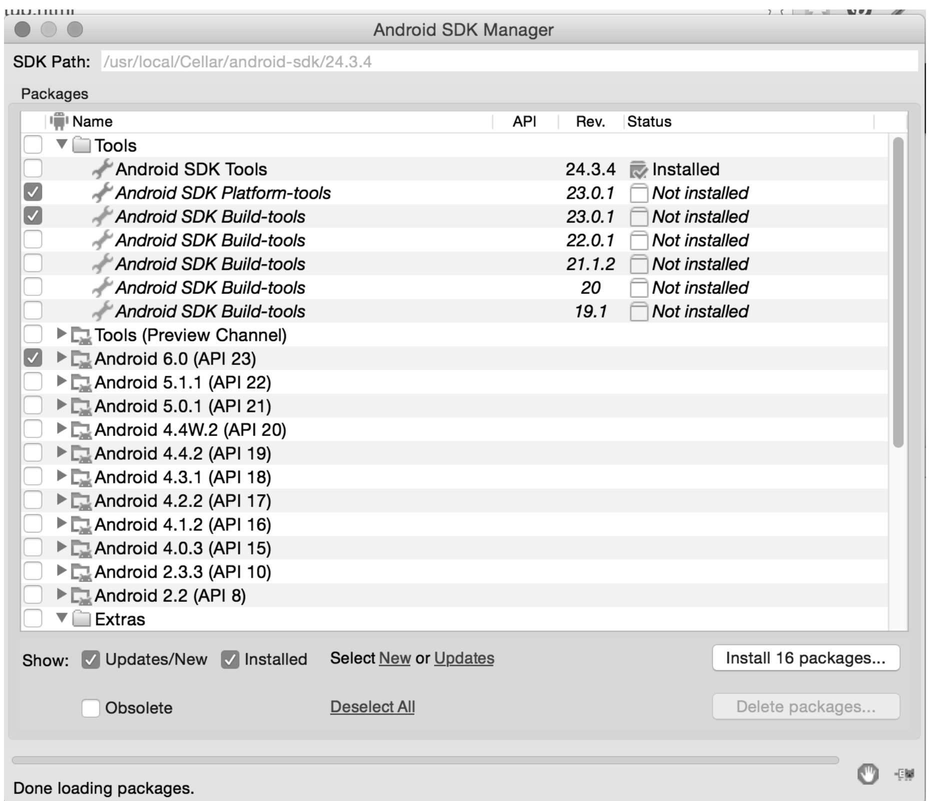


图 3-1：Android SDK 管理器允许你选择开发包进行安装

等待 SDK 管理更新并下载开发包列表。部分开发包会被默认选中，另外要确保选中了以下选项：

- Android SDK Build-tools version 23.0.1
- Android 6.0 (API 23)
- Android Support Repository

然后，点击 Install Packages 并接受合适的许可。等待安装完成可能会花费一些时间。

接下来，你将要安装模拟器和相关的工具。

启动一个新的 shell，然后再次运行 `android` 来启动 Android SDK 管理器。我们将安装一些其他的包。

- Intel x86 Atom System Image (for Android 5.1.1–API 22)
- Intel x86 Emulator Accelerator (HAXM installer)

再次点击 Install Packages，接受合适的许可。

这些依赖包使我们能够创建 Android 虚拟设备（Android Virtual Devices，AVDs）或模拟器，但实际上我们还未创建任何模拟器。让我们来创建它，运行如下命令启动 AVD 管理器（如图 3-2 所示）：

```
android avd
```

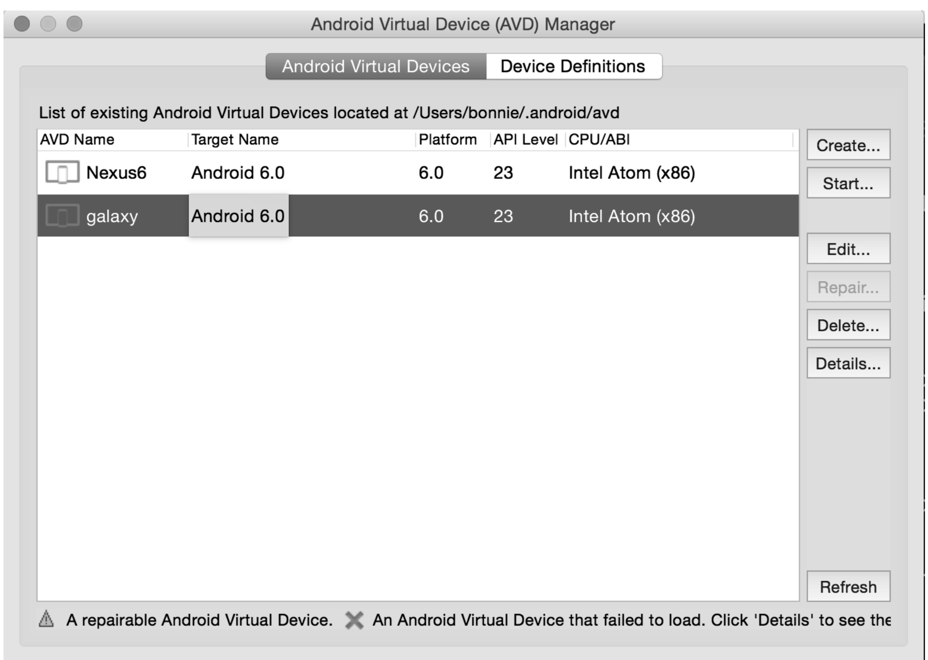


图 3-2：通过 AVD 管理器创建和运行模拟器

之后，点击 Create 按钮并填写创建模拟器的相关信息（如图 3-3 所示）。对于模拟器选项，记得勾选 Use Host GPU（如图 3-4 所示）。

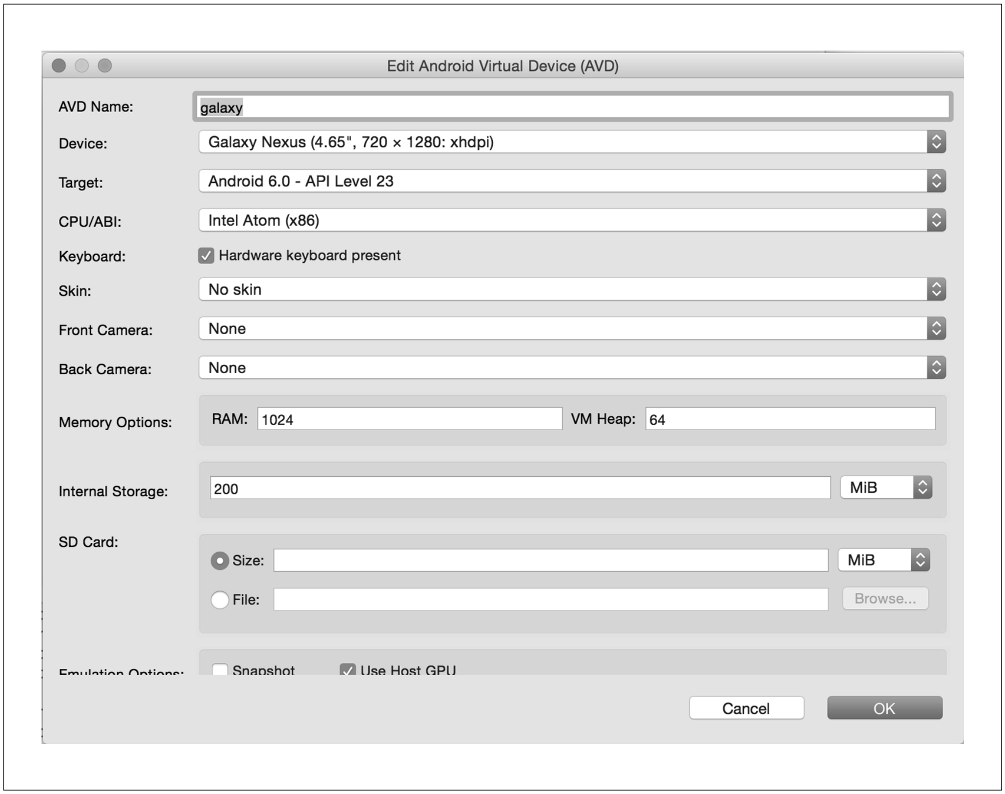


图 3-3：创建任何你喜欢的模拟器（此处创建了一个 Galaxy Nexus 模拟器）

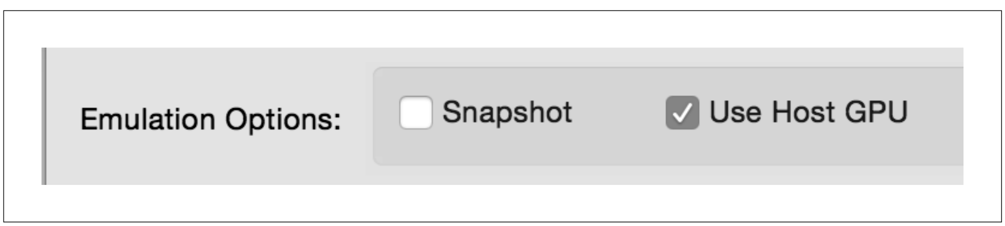


图 3-4：确保已经勾选了 Use Host GPU，否则模拟器会非常慢

如果愿意的话，你可以创建许多 AVD。由于 Android 设备种类繁多，有不同的屏幕尺寸、分辨率和功能，因此使用不同的模拟器通常能为测试带来帮助。当然，出于学习的目的，我们只需要安装一个即可。

3.2　创建一个新的应用

你可以使用 React Native 命令行工具来创建一个新的应用，它会为你生成一个包含 React Native、iOS 和 Android 的全新模板工程：

```
react-native init FirstProject
```

成功创建之后的项目结构如图 3-5 所示。

```
.
├── android
│   ├── app
│   ├── build.gradle
│   ├── gradle
│   ├── gradle.properties
│   ├── gradlew
│   ├── gradlew.bat
│   └── settings.gradle
├── index.android.js
├── index.ios.js
├── ios
│   ├── FirstProject
│   ├── FirstProject.xcodeproj
│   ├── FirstProjectTests
│   └── main.jsbundle
├── node_modules
│   └── react-native
└── package.json
```

图 3-5：默认工程的文件结构

图中 ios/ 和 android/ 目录包含了平台相关的开发模板。你的 React 代码被放在 index.ios.js 和 index.android.js 文件中，它们分别是各自平台的入口文件。通过 npm 安装的依赖文件通常会被放在 node_modules/ 目录下。

如果需要，可以从 GitHub 仓库中下载本书的示例工程（https://github.com/bonniee/learning-react-native）。

3.2.1　在iOS平台运行React Native应用

作为初学者，我们将分别尝试在模拟器和物理设备上运行 iOS 版本的 React Native 应用。

使用 Xcode 打开 ios/ 目录下的 FirstProject.xcodeproj 文件。你会注意到左上方有一个“运行”按钮，如图 3-6 所示。点击“运行”按钮，程序将会在编译之后启动。你也可以选择不同的 iOS 模拟器作为部署目标。



图 3-6：“运行”按钮和部署目标的切换

点击“运行”按钮之后，React 包管理器将会自动运行在新的终端窗口中，如果运行失败或输出错误，请在 FirstProject 目录下重新运行 `npm install` 和 `npm start` 命令。

终端窗口如图 3-7 所示。

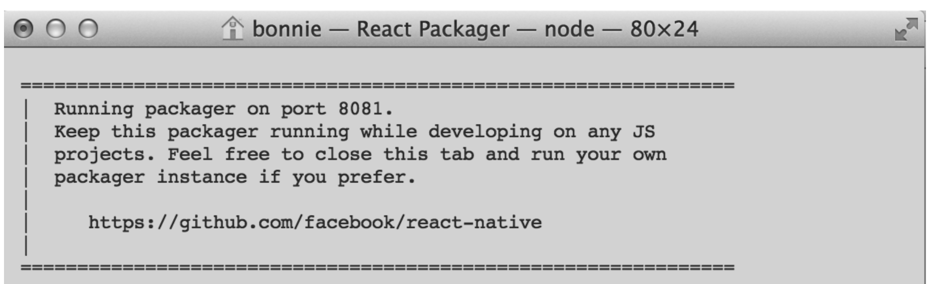


图 3-7：React 包管理器

包管理器就绪之后，iOS 模拟器将会运行默认的应用程序。不出意外的话，结果应如图 3-8 所示。

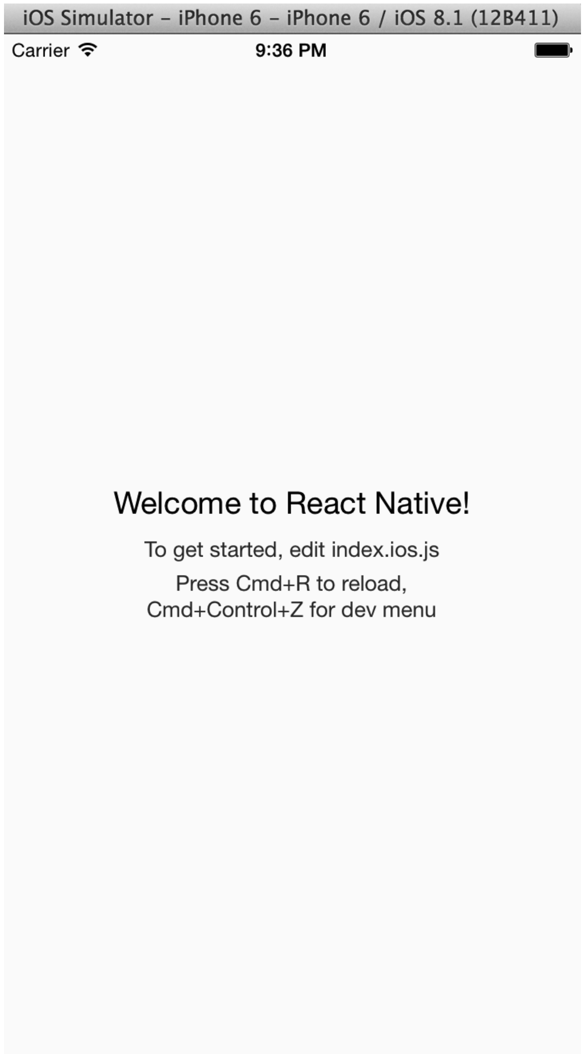


图 3-8：默认应用的截图

为了让代码能在模拟器上实时更新，需要保证包管理器一直处于运行状态。如果包管理器不幸崩溃退出，可切换到工程目录，然后运行 **`npm start`** 命令来重启它。

3.2.2　部署到iOS设备

为了将你的 React Native 应用上传至物理 iOS 设备中，你需要一个 Apple 开发者账号。然后，需要生成证书并注册你的设备。最后，打开 Xcode 偏好设置，添加你的账号即可（如图 3-9 所示）。

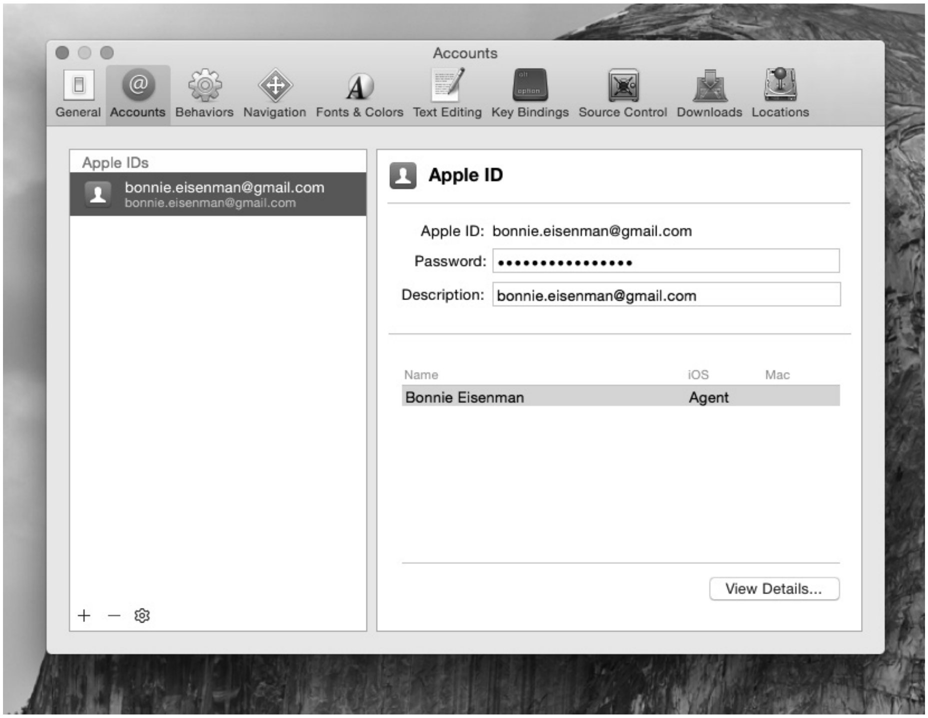


图 3-9：在 Xcode 偏好设置面板添加你的账户

接下来看如何为你的账号生成证书。最简单的办法就是在 Xcode 中打开通用面板（General），如图 3-10 所示，你会看到一个警告的符号。点击 Fix Issue（修复问题）按钮来解除警告。为了从 Apple 公司获取证书，Xcode 将会一步步引导你。

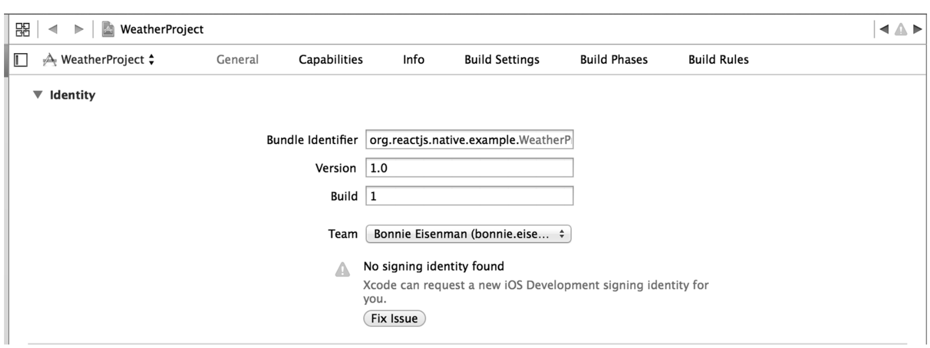


图 3-10：Xcode 通用面板

成功获取证书之后，工作基本就完成了。最后一步是登录到 Apple 开发者中心（http://developer.apple.com），然后注册你的设备（如图 3-11 所示）。

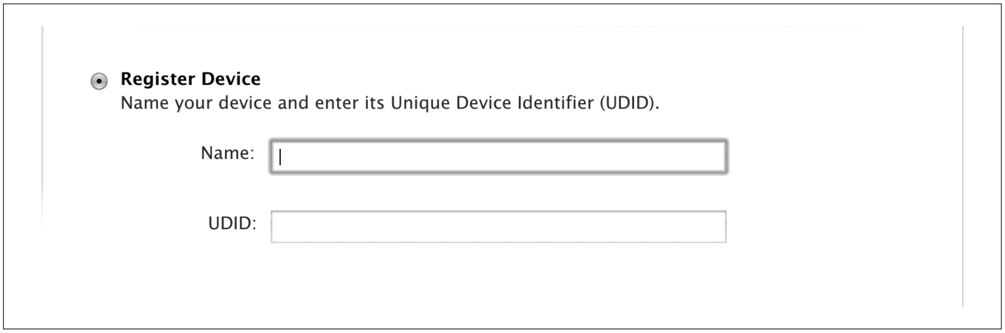


图 3-11：在 iOS 开发者中心注册你的 iOS 设备

设备的 UDID 很容易获取。将你的设备连接到电脑上，打开 iTunes，选择你的设备，然后单击序列号，就可以看到 UDID 显示出来并且复制到剪贴板中了。

一旦将你的设备注册到 Apple 公司之后，构建列表中就会出现许可的设备了。

如果你只想发布到测试设备上，那么这个注册过程也可以在今后进行。另外，Apple 公司每年会通过开发者计划为独立开发者分配 100 个设备用于测试。

最后，我们在部署之前需要对代码作一些改动。你需要在 AppDelegate.m 文件中将 localhost 改成你的 Mac 的 IP 地址。假如你不知道如何查看自己电脑的 IP 地址，可以在终端运行 **`ifconfig`**，在 `en0` 下的 `inet` 即为 IP 地址。

例如，你的 IP 地址为 10.10.12.345，那么应该把 `jsCodeLocation` 修改为：

```
jsCodeLocation =
[NSURL URLWithString:@"http://10.10.12.345:8081/index.ios.bundle"];
```

呀！一路的配置终于完成了，现在我们可以在 Xcode 左上方选择部署的物理设备了（图 3-12）。

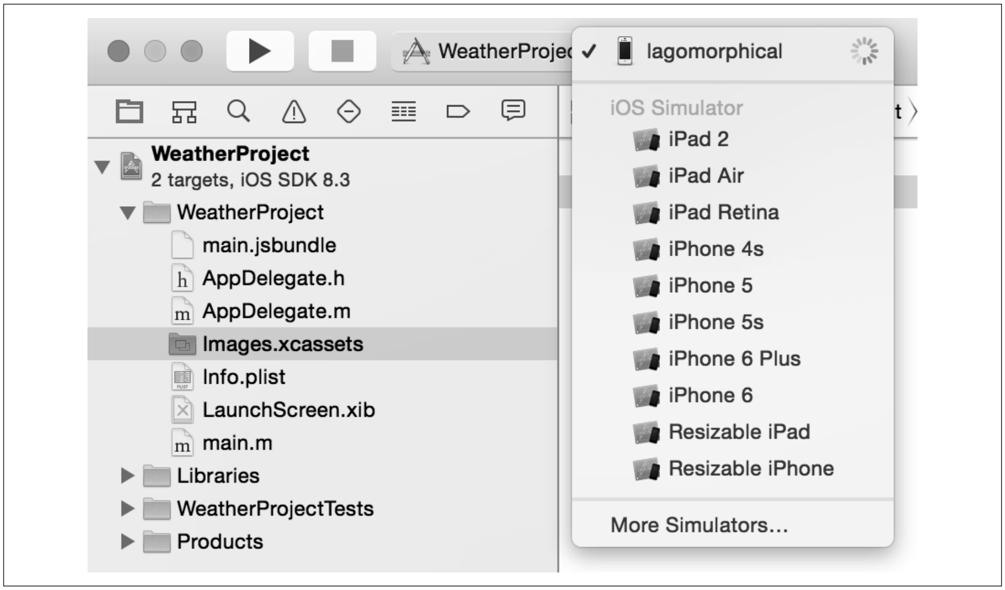


图 3-12：选择你的 iOS 设备作为部署平台

选好之后，单击“运行”按钮，应用程序就安装到你的设备中了，就像在模拟器上一样。你会发现关闭应用程序之后，它已经被安装在主菜单中了。

3.2.3 在Android平台运行React Native应用

为了在 Android 平台运行 React Native 应用，需要做两件事情：首先打开模拟器，然后运行程序。

之前我们介绍过了运行 AVD 管理器的方法（如图 3-2 所示）：

```
android avd
```

选择希望运行的模拟器版本，然后点击 Start... 按钮。

另外，也可以通过命令行来运行模拟器。通过以下命令显示出所有可用的模拟器类型：

```
emulator -list-avds
```

然后通过名字和 @ 前缀来运行它们，例如，我有一个名为 galaxy 的模拟器，我可以这样来运行它：

```
emulator @galaxy
```

无论采用何种方式来启动模拟器，一旦启动成功，只需要在工程的根目录运行如下命令即可加载 React Native 应用：

```
react-native run-android
```

3.2.4　小结：创建并运行项目

以上文字涉及很多知识，因为我们需要为 React Native 安装 iOS 或 Android 设备的各种依赖，看上去比较麻烦。

好消息是，现在你已经完成了初始阶段的这些琐碎的工作，接下来就会变得更轻松。在 React Native 中创建“Hello World”，只需要在命令行运行 **`react-native init HelloWorld`** 即可。

3.3　探索示例代码

我们已经部署并运行了默认的应用程序，接下来看看它是如何工作的。在这一节中，我们将深入到默认应用的源代码中去探索 React Native 项目的结构。

3.3.1　添加组件到视图中

当 React Native 应用启动之后，React 组件是如何被添加到视图中的呢？是什么决定了组件的渲染情况？

问题的答案取决于平台。我们先来看项目的 iOS 版本。

我们可以在 AppDelegate.m 文件中找到答案。例 3-1 尤其要注意。

例 3-1：在 ios/AppDelegate.m 中声明根视图

```
RCTRootView *rootView =
  [[RCTRootView alloc] initWithBundleURL:jsCodeLocation
  moduleName:@"FirstProject"
  launchOptions:launchOptions];
```

React Native 库将其所有的类名使用 `RCT` 作为前缀，也就是说 `RCTRootView` 就是一个 React Native 类。所以，`RCTRootView` 代表 React Native 的根视图。AppDelegate.m 中其他的代码则将视图添加到 `UIViewController` 中并渲染到屏幕上。这个步骤与使用 `React.render` 方法挂载 React 组件到 DOM 节点上有着异曲同工之妙。

眼下，AppDelegate.m 文件有两处应该修改。

第一处需要修改的地方是 `jsCodeLocation` 这一行，之前为了把应用部署到物理设备上，我们修改过此处。正如代码中的注释所示，第一种方式作为开发使用，第二种方式用来将预打包文件部署到硬盘上。现在我们采用第一种方式，今后一旦需要部署到应用商店，我们会更加详细地讨论这两种方式。

另一处需要修改的地方是 moduleName，它被传递给 RCTRootView 以决定哪个组件将被挂载到视图中。这里你可以指定哪些组件需要被程序渲染。

为了使用 FirstProject 组件，你需要在 React 中注册一个相同名字的组件。如果打开 index.ios.js，你会看到代码的最后一行已经完成了这项工作（例 3-2）。

例 3-2：注册顶层组件

```
AppRegistry.registerComponent('FirstProject', () => FirstProject);
```

以上代码暴露了 FirstProject 组件，使得我们能够在 AppDelegate.m 文件中使用它。大多数情况下，你都不需要去修改这个模板代码，但是我们应该对此有一些了解。

那么，Android 平台是怎样的呢？原理也很类似。如果你查看 MainActivity.java 文件，会注意到这一行代码（例 3-3）。

例 3-3：MainActivity.java 中 Android 的 React 入口

```
mReactRootView.startReactApplication(mReactInstanceManager, "FirstProject", null);
```

正如 AppDelegate.m 之于 iOS，Android 的 MainActivity.java 也会查看 AppRegistry 中绑定到 FirstProject 的 React 组件。

3.3.2 React Native中的模块导入

让我们进一步观察 index.ios.js 文件。如例 3-4 所示，require 语句跟通常的用法有一些区别。

例 3-4：React Native 中的 require 语句，导入 UI 元素

```
var React = require('react-native');
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
} = React;
```

上面的语法挺有趣的。我们通过 require 语句导入了 React，但是下一行代码发生了什么呢？

React Native 的使用方面有一点比较奇特，那就是你要导入所需的每一个组件或模块。诸如 <div> 之类的标签是不存在的，如果你需要使用 <View> 和 <Text> 等组件，就要逐一导入。像 Stylesheet 和 AppRegistry 这样的库函数也需要使用以上语法进行导入。一旦开始开发自己的应用，我们将会探索 React Native 提供的其他库函数，同样也是需要导入才能使用。

如果你不熟悉这些语法，可以在附录 A 中查看例 A-2，它解释了 ES6 的解构特性。

3.3.3　FirstProject组件

让我们看看 `<FirstProject>` 组件（例 3-5），代码存在于 index.ios.js 和 index.android.js 文件中（它们是相同的）。

这些代码看起来亲切而熟悉，因为 `<FirstProject>` 仅仅是一个普通的 React 组件，主要的不同在于用 `<Text>` 和 `<View>` 代替了 `<div>` 和 `<span>`，并且用对象来表示样式。

例 3-5：包含样式的 FirstProject 组件

```
var FirstProject = React.createClass({
  render: function() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to React Native!
         </Text>
         <Text style={styles.instructions}>
           To get started, edit index.ios.js
         </Text>
         <Text style={styles.instructions}>
           Press Cmd+R to reload,{'\n'}
           Cmd+D or shake for dev menu
         </Text>
       </View>
     );
   }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  instructions: {
    textAlign: 'center',
    color: '#333333',
    marginBottom: 5,
  },
});
```

正如之前所提到的，React Native 中所有的样式都采用样式对象来代替传统的样式表，标准的做法就是利用 `StyleSheet` 库进行样式的编写。你可以在文件的底部看到样式对象是如何定义的。需要注意的是，`<Text>` 组件可以使用文本特有的属性，如 `fontSize`，所有的布局样式都使用 flexbox。我们将在第 5 章用更多篇幅介绍布局的内容。

该示例应用很好地介绍了创建 React Native 应用的一些基本函数。它挂载了 React 组件用于渲染，并介绍了 React Native 的基本样式和渲染逻辑，同时也可以检验我们的开发环境是否被正确安装，我们还尝试将应用部署到真实设备上。但它依然是一个极其基础的缺乏用户交互的应用，接下来让我们尝试开发一个有更多功能的应用吧。

3.4 开发天气应用

我们将使用示例程序来开发天气应用（你也可以通过 **`react-native init WeatherProject`** 创建一个新的示例工程）。这个项目包括如何利用和结合样式表、flexbox、网络通信、用户输入和图像显示等知识来开发一个实用的应用程序，然后将其部署到 Android 或 iOS 设备上。

这部分内容可能会有些含糊不清，因为我们主要把精力集中在这些特性的用法上，而不是深入分析它们。不用担心进度太快，在后续的章节中，我们会将这个天气应用作为参考并深入讨论这些特性。

天气应用最终的界面如图 3-13 所示，用户可以通过文本框输入邮编进行查询。该应用利用 OpenWeatherMap 的接口获取数据并展现当前天气情况。

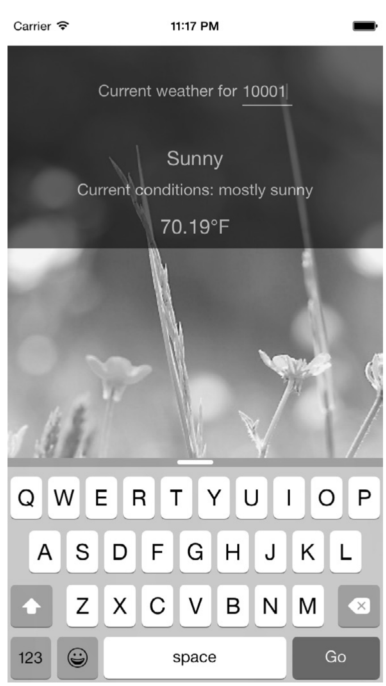


图 3-13：天气应用成品

我们要做的第一件事就是替换默认的代码。将初始组件的代码移动到 WeatherProject.js 中，并修改 index.ios.js 和 index.android.js 文件的内容（例 3-6）。

例 3-6：精简后的 index.ios.js 和 index.android.js 代码（二者保持一致）

```
var React = require('react-native');
var { AppRegistry } = React;
var WeatherProject = require('./WeatherProject');
AppRegistry.registerComponent('WeatherProject', () => WeatherProject);
```

3.4.1　处理用户输入

我们希望用户通过输入邮编获取该地区的天气预报，因此需要添加一个输入框提供给用户。首先，添加默认邮编信息至组件的初始状态（`state`）中（例 3-7）。

例 3-7：在 `render` 函数前加入这段代码

```
getInitialState: function() {
  return {
    zip: ''
  };
}
```

记住，`getInitialState` 可以让我们创建 React 组件的初始状态（`state`）。如果你需要复习 React 组件的生命周期，可以查看 React 文档（https://facebook.github.io/react/docs/component-specs.html）。

接着，修改其中一个 `<Text>` 组件的内容为 `this.state.zip`：

```
<Text style={styles.welcome}>
  You input {this.state.zip}.
</Text>
```

我们用同样的方式添加一个 `<TextInput>` 组件（这是一个允许用户输入文本的基础组件）。

```
<TextInput
  style={styles.input}
  onSubmitEditing={this._handleTextChange}/>
```

这个 `<TextInput>` 组件的文档和属性可以在 React Native 官网查看（http://facebook.github.io/react-native/docs/textinput.html）。为了监听一些事件，你可以往 `<TextInput>` 中传入回调函数，如 `onChange` 和 `onFocus`，但现在暂时不需要这么做。

注意，我们已经为其添加了样式，`input` 样式表如下：

```
var styles = StyleSheet.create({
  ...
  input: {
    fontSize: 20,
    borderWidth: 2,
```

```
      height: 40
      }
    ...
  });
```

在 <TextInput> 组件的属性（prop）中监听了 onSubmitEditing 事件，事件回调需要作为组件中的一个函数：

```
_handleTextChange(event) {
 console.log(event.nativeEvent.text);
 this.setState({zip: event.nativeEvent.text})
}
```

图中 console 语句是外部的，如果需要的话，可以在调试工具里使用该语句进行调试。

同时需要更新导入语句：

```
var React = require('react-native');
var {
  ...
  TextInput
  ...
} = React;
```

现在，尝试使用 iOS 模拟器运行你的应用。它可能不是很美观，但你应该可以成功地输入一个邮编并显示在 <Text> 组件上。

如果需要的话，也可以对用户的输入做一个验证来确保正确输入了五位数字，现在暂时略过。

例 3-8 展示了 WeatherProject.js 组件的完整代码。

例 3-8：WeatherProject.js：这个版本简单地接收并记录用户的输入

```
var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  TextInput,
  Image
} = React;

var WeatherProject = React.createClass({
  // 如果你需要一个默认的邮编，你可以在这里添加一个。
  getInitialState() {
    return ({
      zip: ''
    });
  },
  // 我们将添加这个回调函数到<TextInput>属性中。
  _handleTextChange(event) {
```

```
    // log语句输出结果在Xcode或Chrome调试工具中可见。
    console.log(event.nativeEvent.text);

    this.setState({
      zip: event.nativeEvent.text
    });
  },
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          You input {this.state.zip}.
        </Text>
        <TextInput
              style={styles.input}
              onSubmitEditing={this._handleTextChange}/>
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  input: {
    fontSize: 20,
    borderWidth: 2,
    height: 40
    }
});

module.exports = WeatherProject;
```

3.4.2　展现数据

现在，我们来开发根据邮编查询天气预报的功能。首先添加一些 mock 数据（虚拟的数据）到 WeatherProject.js 文件的 `getInitialState` 方法中。

```
getInitialState() {
  return {
    zip: '',
    forecast: {
      main: 'Clouds',
```

```
        description: 'few clouds',
        temp: 45.7
      }
    }
  }
```

为了让程序更加清晰，我们把天气预报独立成一个单独的组件，新建一个名为 Forecast.js 的文件（例 3-9）。

例 3-9：Forecast.js 中的 Forecast 组件

```
var React = require('react-native');
var {
  StyleSheet,
  Text,
  View
} = React;

var Forecast = React.createClass({
  render: function() {
    return (
      <View>
        <Text style={styles.bigText}>
          {this.props.main}
        </Text>
        <Text style={styles.mainText}>
          Current conditions: {this.props.description}
        </Text>
        <Text style={styles.bigText}>
          {this.props.temp}°F
        </Text>
      </View>
    );
  }
});


var styles = StyleSheet.create({
  bigText: {
    flex: 2,
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
    color: '#FFFFFF'
  },
  mainText: {
    flex: 1,
    fontSize: 16,
    textAlign: 'center',
    color: '#FFFFFF'
  }
})

module.exports = Forecast;
```

`<Forecast>` 组件只能基于它的属性渲染一些 `<Text>` 文本，我们也会在文件的末尾添加一些简单的文本颜色之类的样式。

导入 `<Forecast>` 组件并添加到 `render` 方法中，通过 `this.state.forecast` 向它的属性传入数据（例 3-10）。我们稍后会解决布局和样式问题。你能在图 3-14 看到 `<Forecast>` 组件最终显示的效果。

例 3-10：WeatherProject.js 中应该加入新的 `state` 和 Forecast 组件

```
var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  TextInput,
  Image
} = React;

var Forecast = require('./Forecast');

var WeatherProject = React.createClass({
  getInitialState() {
    return {
      zip: '',
      forecast: {
        main: 'Clouds',
        description: 'few clouds',
        temp: 45.7
      }
    }
  },
  _handleTextChange(event) {
    console.log(event.nativeEvent.text);
    this.setState({
      zip: event.nativeEvent.text
    });
  },
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          You input {this.state.zip}.
        </Text>
        <Forecast
          main={this.state.forecast.main}
          description={this.state.forecast.description}
          temp={this.state.forecast.temp}/>
        <TextInput
          style={styles.input}
          returnKeyType='go'
          onSubmitEditing={this._handleTextChange}/>
      </View>
    );
```

```
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#4D4D4D',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  input: {
    fontSize: 20,
    borderWidth: 2,
    height: 40
    }
});

module.exports = WeatherProject;
```

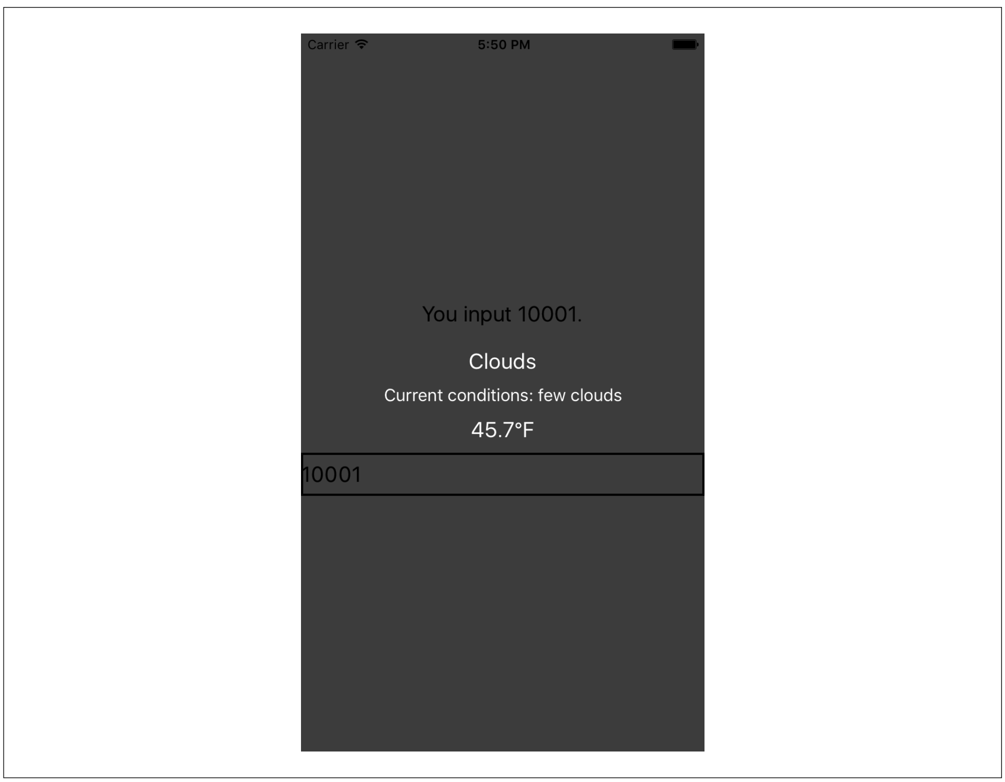


图 3-14：目前的天气应用

3.4.3 添加背景图片

单纯的背景颜色略显单调。我们接下来为其添加一个背景图片。[1]



资源导入的方法是平台特定的

Android 和 iOS 添加资源的方法是不同的。这里我们讲解两种方法。

像图片这样的资源需要根据平台分别导入到你的项目工程中。我们先看看 Xcode 如何处理。

选择 Images.xcassets/ 目录，然后选择 New Image Set 选项，如图 3-15 所示。然后，你可以拖放一张图片到这个图片集中，图 3-16 展示了最终的效果。要确保图片集的名字与文件名保持一致，否则 React Native 可能无法导入。

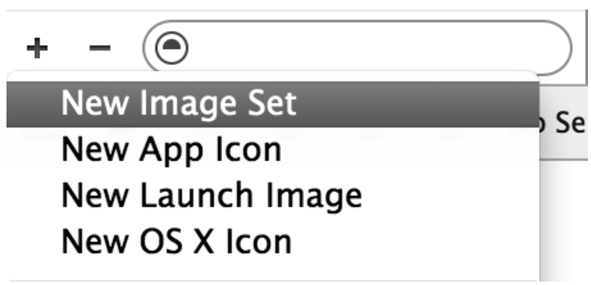


图 3-15：新建一个图片集

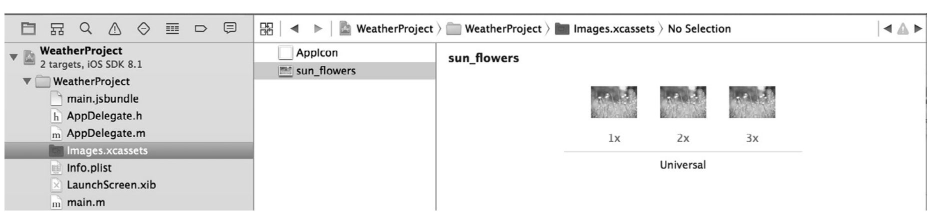


图 3-16：拖放图片至图片集中

@2x 和 @3x 修饰符分别表明图片显示在基准分辨率两倍或三倍的屏幕中。由于我们的天气应用是一个通用的软件（意味着一个程序可以同时运行在 iPhone 或 iPad 中），Xcode 赋予了我们针对不同分辨率添加不同图片的能力。

注 1：React Native 0.14 版本之后，提供了图片统一的管理方式，详见 http://facebook.github.io/react-native/docs/images.html。以下篇幅介绍的是 React Native 0.14 之前版本的做法，如使用最新版本可略过该部分。——译者注

对于 Android 系统，我们需要将图片文件作为可绘制位图资源（bitmap drawable resources）（http://developer.android.com/guide/topics/resources/drawable-resource.html#Bitmap）添加到目录 WeatherProject/android/app/src/main/res 中。你需要将 .png 图片添加到下列特定分辨率的目录中（如图 3-17 所示）：

- drawable-mdpi/ (1x)
- drawable-hdpi/ (1.5x)
- drawable-xhdpi/ (2x)
- drawable-xxhdpi/ (3x)

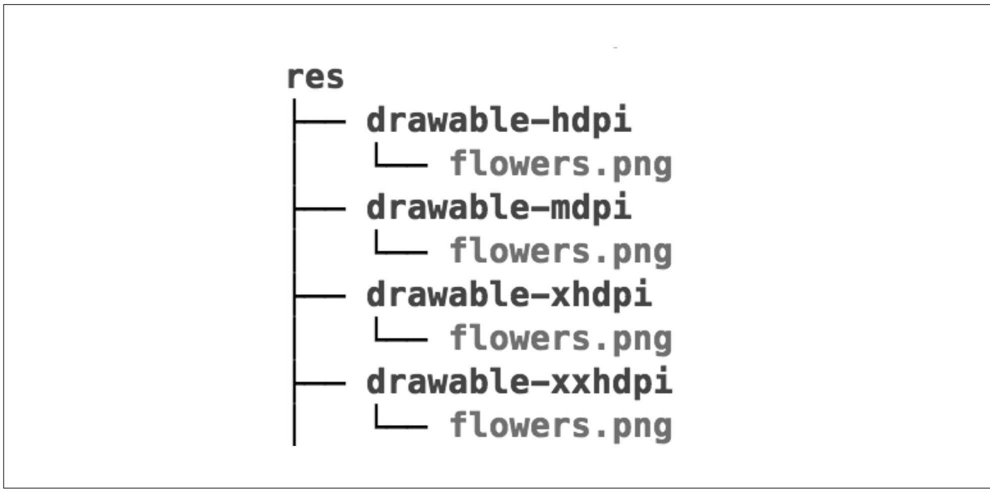


图 3-17：添加图片文件到 Android 中

之后，图片就可以在 Android 应用中使用了。

这个工作流程可能让你感觉不舒服，这就是它的现状。不过它很可能在之后版本的 React Native 中进行改进。

既然文件已经被导入到 Android 和 iOS 项目中，让我们回到 React 代码吧。添加一背景图片时，我们不能像在 Web 环境一样为 `<div>` 标签设置属性，而是将 `<Image>` 组件作为容器使用：

```
<Image source={require('image!flowers')}
       resizeMode='cover'
       style={styles.backdrop}>
  // 这里放置内容。
</Image>
```

`<Image>` 组件期望一个图片源 `prop`，我们通过 `require` 进行引入。`require(image!flowers)` 语句将会触发 React Native 查询名为 flowers 的文件。

别忘了为其样式添加 flexDirection 属性，使得它的子元素渲染成我们期望的样子：

```
backdrop: {
  flex: 1,
  flexDirection: 'column'
}
```

现在为 <Image> 添加一些子元素。更新 <Weather Project> 组件的 render 方法为：

```
<Image source={require('image!flowers')}
       resizeMode='cover'
         style={styles.backdrop}>
    <View style={styles.overlay}>
      <View style={styles.row}>
        <Text style={styles.mainText}>
          Current weather for
        </Text>
        <View style={styles.zipContainer}>
          <TextInput
            style={[styles.zipCode, styles.mainText]}
            returnKeyType='go'
            onSubmitEditing={this._handleTextChange}/>
        </View>
      </View>
      <Forecast
         main={this.state.forecast.main}
         description={this.state.forecast.description}
         temp={this.state.forecast.temp}/>
    </View>
  </Image>
```

你会发现我使用了一些还没有讨论过的样式，例如：row、overlay、zipContainer 和 zipCode 等样式。你可以跳到这部分的末尾去查看完整的样式表。

3.4.4 从Web获取数据

下一步，我们将探索 React Native 中网络接口的用法。你不能在移动设备中使用 jQuery 发送 AJAX 请求。然而，React Native 实现了 Fetch 接口。基于 Promise 的语法非常简洁：

```
fetch('http://www.somesite.com')
  .then((response) => response.text())
  .then((responseText) => {
    console.log(responseText);
  });
```

我们将会使用 OpenWeatherMap 的接口，它提供给我们一个可以根据邮编返回当前天气情况的简单的端点。

为了集成这个接口，我们可以修改 <TextInput> 组件的回调函数，从而使用 OpenWeatherMap 的接口进行查询。

```
_handleTextChange: function(event) {
  var zip = event.nativeEvent.text;
  this.setState({zip: zip});
  fetch('http://api.openweathermap.org/data/2.5/weather?q=' +
  zip + '&units=imperial')
    .then((response) => response.json())
    .then((responseJSON) => {
      // 如果你愿意,可以看看这个格式。
      console.log(responseJSON);
      this.setState({
        forecast: {
          main: responseJSON.weather[0].main,
          description: responseJSON.weather[0].description,
          temp: responseJSON.main.temp
        }
      });
    })
    .catch((error) => {
      console.warn(error);
    });
}
```

注意，我们需要从返回结果中获取 JSON。使用 Fetch 接口是非常直观的，以上就是我们需要做的全部工作了。

另一件我们要做的事是移除占位的数据，确保在没有数据的时候，天气预报组件不会被渲染。

首先，从 `getInitialState` 中清除 mock 数据：

```
getInitialState: function() {
  return {
    zip: '',
    forecast: null
  };
}
```

然后，在 `render` 函数中更新渲染逻辑：

```
var content = null;
if (this.state.forecast !== null) {
  content = <Forecast
              main={this.state.forecast.main}
              description={this.state.forecast.description}
              temp={this.state.forecast.temp}/>;
}
```

最后，在 `render` 函数中使用 `{content}` 替换 `<Forecast>` 组件。

3.4.5 整合

对于这个应用的最后一个版本，我重新组织了 `<WeatherProject>` 组件的 `render` 函数并且

调整了样式。最大的改变是布局逻辑，如图 3-18 所示。

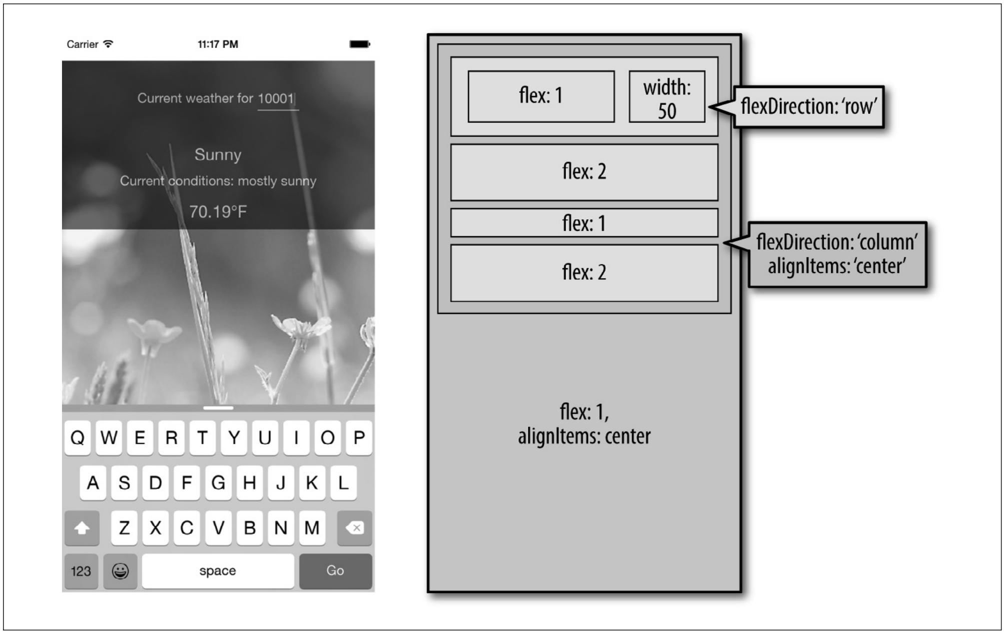


图 3-18：天气应用最终的布局

好，准备好查看整体代码了吗？例 3-11 展示了完成之后的 `<WeatherProject>` 组件包括样式表在内的完整代码。`<Forecast>` 组件仍然与例 3-9 一致。

例 3-11：WeatherProject.js 完整代码

```
var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  TextInput,
  Image
} = React;
var Forecast = require('./Forecast');

var WeatherProject = React.createClass({
  getInitialState: function() {
    return {
      zip: '',
      forecast: null
    };
  },

  _handleTextChange: function(event) {
    var zip = event.nativeEvent.text;
    this.setState({zip: zip});
```

```
    fetch('http://api.openweathermap.org/data/2.5/weather?q='
      + zip + '&units=imperial')
      .then((response) => response.json())
      .then((responseJSON) => {
        this.setState({
          forecast: {
            main: responseJSON.weather[0].main,
            description: responseJSON.weather[0].description,
            temp: responseJSON.main.temp
          }
        });
      })
      .catch((error) => {
        console.warn(error);
      });
  },

  render: function() {
    var content = null;
    if (this.state.forecast !== null) {
      content = <Forecast
                  main={this.state.forecast.main}
                  description={this.state.forecast.description}
                  temp={this.state.forecast.temp}/>;
    }
    return (
      <View style={styles.container}>
        <Image source={require('image!flowers')}
               resizeMode='cover'
               style={styles.backdrop}>
          <View style={styles.overlay}>
           <View style={styles.row}>
             <Text style={styles.mainText}>
               Current weather for
             </Text>
             <View style={styles.zipContainer}>
               <TextInput
                 style={[styles.zipCode, styles.mainText]}
                 returnKeyType='go'
                 onSubmitEditing={this._handleTextChange}/>
             </View>
           </View>
           {content}
          </View>
        </Image>
      </View>
    );
  }
});

var baseFontSize = 16;
var styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
```

```
    paddingTop: 30
  },
  backdrop: {
    flex: 1,
    flexDirection: 'column'
  },
  overlay: {
    paddingTop: 5,
    backgroundColor: '#000000',
    opacity: 0.5,
    flexDirection: 'column',
    alignItems: 'center'
  },
  row: {
    flex: 1,
    flexDirection: 'row',
    flexWrap: 'nowrap',
    alignItems: 'flex-start',
    padding: 30
  },
  zipContainer: {
    flex: 1,
    borderBottomColor: '#DDDDDD',
    borderBottomWidth: 1,
    marginLeft: 5,
    marginTop: 3
  },
  zipCode: {
    width: 50,
    height: baseFontSize,
  },
  mainText: {
    flex: 1,
    fontSize: baseFontSize,
    color: '#FFFFFF'
  }
});

module.exports = WeatherProject;
```

我们已经完成了所有的工作，现在尝试运行这个应用。不出意外的话，它将可以同时运行在 Android 和 iOS 设备上，无论是模拟器还是真实物理设备皆可。想对它进行修改或者完善吗？

你可以在 GitHub 仓库查看完整的代码（https://github.com/bonniee/learning-react-native）。

3.5　小结

我们开发的第一个应用涉及了很多知识。本章介绍了新的 UI 组件——`<TextInput>`，以及如何从中获取用户输入的信息。接下来本章解释了如何在 React Native 中编写基本样式，以及如何在应用中加载并使用图片。最后本章讨论了如何使用 React Native 网络接口从外

部网络源中请求数据。对于我们的第一个应用，这已经相当不错了！

幸运的是，这足以证明你可以使用 React Native 快速地开发出具有原生体验的、功能丰富的移动应用。

如果你想继续扩展这个应用，可以尝试：

- 添加更多的图片，并根据天气预报更换图片；
- 对邮编进行有效性验证；
- 切换更便捷的小型键盘进行邮编输入；
- 展示最近 5 天的天气预报。

随着我们学习更多的知识，例如地理位置，你将可以为天气应用扩展更多的功能。

当然，这只是一个快速的概览。在后面几章中，我们将更加深入地了解 React Native 的最佳实践，并学习使用更多的特性！

第 4 章

移动应用组件

在第 3 章中，我们构建了一个简单的天气应用，并从中学习了创建 React Native 界面的基础知识。在本章中，我们将更深入地了解 React Native 的移动应用组件，并将其与基础 HTML 元素进行比较。相比于网页，移动界面是基于各种不同的原始 UI 元素而构造的，因此我们需要使用不同的组件。

在本章的开头，我们将详细了解一些最基础的组件：`<View>`、`<Image>` 和 `<Text>`，然后讨论触摸和手势怎样作用于 React Native 组件，并了解如何处理触摸事件。紧接着我们学习一些更高阶的组件，例如 `<ListView>`、`<TabView>` 和 `<NavigatorView>`，让你可以组合这些视图开发出符合界面规范的移动应用。

4.1　类比HTML元素与原生组件

当开发 Web 应用时，我们使用各种基础的 HTML 元素，例如 `<div>`、`<span>` 和 `<img>`，还有各种组织元素：`<ol>`、`<ul>` 和 `<table>`。（也可以考虑像 `<audio>`、`<svg>`、`<canvas>` 等元素，但此处暂时忽略它们。）

在 React Native 中，我们不使用这些 HTML 元素，但使用类似于它们的各种组件（表 4-1）。

表4-1：类比HTML和原生组件

HTML	React Native
div	View
img	Image
span, p	Text
ul/ol, li	ListView, child items

虽然这些元素有相似的作用，但它们不可以相互替换。让我们来看这些组件是如何在 React Native 的应用中工作的，并了解它们与浏览器场景有什么区别。

> **React Native 和 Web 应用的代码可以共享吗？**
>
> 很遗憾，React Native 基础组件目前不能渲染成基础的 HTML 元素。你的 React Native 代码可以在 iOS 和 Android 中复用（以及 React Native 未来的其他平台），但它不能渲染出兼容浏览器的视图。然而，包含 React 组件在内的任何不参与基础元素渲染的 JavaScript 代码都可以被复用。因此，如果你的业务逻辑独立于渲染的代码，你可以进行代码复用。[1]

4.1.1 文本组件

渲染文本看似是一个非常基础的功能，几乎所有的应用都需要在某些地方渲染文本。然而，在 React Native 环境和移动开发中的文本渲染与 Web 环境大相径庭。

当我们在 HTML 中处理文本时，可以将原始文本包含在各种元素中，并且还可使用像 <strong> 和 <em> 这样的子标签为文本添加样式。因此，你可以写出下面这样的 HTML 代码：

```
<p>The quick <em>brown</em> fox jumped over the lazy <strong>dog</strong>.</p>
```

在 React Native 中，仅有 <Text> 组件可以将纯文本作为子节点。换言之，这样是无效的：

```
<View>
  Text doesn't go here!
</View>
```

应该把文本用 <Text> 组件包装起来。

```
<View>
  <Text>This is OK!</Text>
</View>
```

在 React Native 中使用 <Text> 组件时，我们再也不能使用 <strong> 和 <em> 这样的子标签，

注 1：目前有 React Native 与 React Web 代码共享的开源解决方案，例如 react-web。——译者注

但是可以应用样式中的 fontWeight 和 fontStyle 等属性来达到相似的效果。以下是使用内联样式达到的效果：

```
<Text>
  The quick <Text style={{fontStyle: "italic"}}>brown</Text> fox
  jumped over the lazy <Text style={{fontWeight: "bold"}}>dog</Text>.
</Text>
```

但是这样的方法很快会让代码变得冗长。处理文本时，你应该会想创建一种便于速记的样式组件，如例 4-1 所示。

例 4-1：为文本样式创建可复用组件

```
var styles = StyleSheet.create({
  bold: {
      fontWeight: "bold"
  },
  italic: {
      fontStyle: "italic"
  }
});

var Strong = React.createClass({
  render: function() {
    return (
    <Text style={styles.bold}>
      {this.props.children}
    </Text>);
  }
});

var Em = React.createClass({
  render: function() {
    return (
    <Text style={styles.italic}>
      {this.props.children}
    </Text>);
  }
});
```

一旦声明了这些样式组件，你就可以自由地使用样式嵌套。现在 React Native 版本已经和 HTML 版本很相似了（例 4-2）。

例 4-2：使用样式组件渲染文本

```
<Text>
  The quick <Em>brown</Em> fox jumped
  over the lazy <Strong>dog</Strong>.
</Text>
```

同样，React Native 也没有 header 元素（h1、h2 等）这样的概念，但在需要的时候很容易定义自己的 <Text> 样式。

总体来说，当你需要应付字体样式问题时，React Native 强制改变你原先的方法。样式的继承是有限的，因此你无法获得渲染树中所有文本节点的默认字体设置。前面已经提到，Facebook 建议通过使用样式组件来解决这个问题：

> 你也失去了为整棵子树设置默认字体的能力。我们推荐在应用中创建并使用一个统一字体和尺寸的 MyAppText 组件，以此来确保字体样式的一致性。你也可以在此基础上为其他的字体样式创建更具体的组件，例如 MyAppHeaderText。
>
> —— React Native 文档

官方文档的文本组件部分（https://facebook.github.io/react-native/docs/text.html#limited-style-inheritance）有更多这方面的细节。

你可能注意到这里的一个模式：React Native 坚持自己的偏好，提倡样式组件的复用而不是样式的复用。我们将在下一章更深入地讨论这些内容。

4.1.2　图片组件

如果说文本是移动或 Web 应用中最基础的元素，那么图片元素也不例外。当在 Web 环境中编写 HTML 和 CSS 时，我们可以通过多种方式添加图片：有时我们使用 `<img>` 标签，有时通过 CSS 导入，如 `background-image` 属性。在 React Native 中，我们也有相似的 `<Image>` 组件，但是它有些不太一样。

`<Image>` 组件的基础用法是很直观的。只需设置 `prop` 的 `source` 属性：[2]

```
<Image source={require('image!puppies')} />
```

这个 `require` 语句是如何工作的呢？这些资源存在何处呢？在 React Native 中，你需要根据目标平台进行调整，这是其中一个例子。在 iOS 平台，意味着你需要通过指定合适的 @2x、@3x 分辨率文件并导入资源文件到 Xcode 项目中，这样才能在 Xcode 中允许通过不同平台选择正确的资源文件。这对于 Web 开发来说是个不错的改进，iOS 相对固定的屏幕尺寸和分辨率的组合使得可以更容易地创建目标资源。

对于其他平台的 React Native 来说，我们期望 `image!` 语法找出资源目录。

值得一提的是，基于 Web 的图片资源可以被导入，而无需打包图片到应用中。Facebook 将以下代码作为 UIExplorer 中的一个示例：

```
<Image source={{uri: 'https://facebook.github.io/react/img/logo_og.png'}}
       style={{width: 400, height: 400}} />
```

注 2：自 React Native 0.14 版本后，改为了更加方便的用法：`<Image source={require ('. / my-icon.png')} / >`。——译者注

当使用网络资源时，需要手动指定图片尺寸。

通过网络下载而不是作为资源导入的方式有一些优点。例如：在开发期间，采用这种方法可以更容易地完成项目原型，而无需提前小心翼翼地导入所有资源。同时，它也减小了应用体积，用户无需下载你所有的资源文件。然而，这意味着今后无论何时用户使用你的应用都要依赖数据流量。因此在多数情况下，你都应该避免使用基于 URI 的方式。

如果你好奇如何使用用户自己的图片，请看第 6 章。

由于 React Native 强调了基于组件的开发方式，因此图片必须包含在 `<Image>` 组件中，而不允许通过样式进行导入。例如在第 3 章中，我们想要在天气应用中使用一张图片作为背景。不像 HTML 和 CSS 那样可以使用 `background-image` 属性来添加背景，React Native 使用 `<Image>` 作为容器组件，就像这样：

```
<Image source={require('image!puppies')}>
  {/* Your content here... */}
</Image>
```

为图片添加样式是相对容易的。除了应用样式，你还可以通过特定的属性控制图片的渲染行为。例如，你可能会经常用到 `resizeMode` 属性，它可以被设置为 `stretch`、`cover` 和 `contain`。UIExplorer 示例程序也很好地解释了这个属性（图 4-1）。

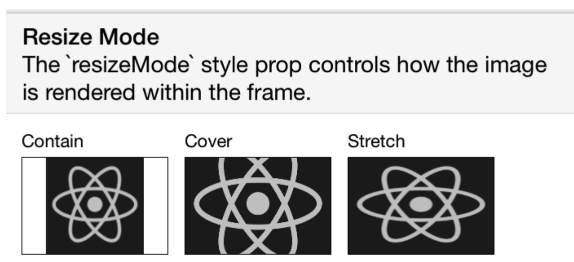


图 4-1：`stretch`、`cover` 和 `contain` 的区别

`<Image>` 组件灵活易用，你可以在自己的应用中广泛使用。

4.2　处理触摸和手势

基于 Web 的界面通常是为鼠标控制而设计的。我们使用如 `hover` 这样的状态来进行动态变换，并对用户的交互行为作出反馈。对于移动应用而言，触摸则至关重要。移动平台对于你想要设计的界面有一套自己的规范，这个规范因平台而不同：iOS 和 Android 不同，Windows Phone 则跟它们又不一样。

React Native 提供了许多 API（应用编程接口），你可以使用这些 API 开发可触摸的界面。在这一节中，我们将要学习 `<TouchableHighlight>` 这个容器组件，由 `PanResponder` 提供的低层 API，以及手势响应系统。

4.2.1 使用TouchableHighlight

任何能响应用户触摸事件的界面元素（按钮、控制元素等）通常都需要用 `<TouchableHighlight>` 包装起来。当视图元素被触摸时，`<TouchableHighlight>` 会产生一个叠层，给予用户视觉反馈。这是其中一个关键性的交互功能，它能让应用具备原生体验，不像那些针对移动优化的网站仅提供有限的触摸反馈。根据我的经验，无论哪里需要一个类似 Web 平台上的按钮或者链接，你都可以使用 `<TouchableHighlight>`。

`<TouchableHighlight>` 最基础的用法就是用它将组件包装起来，之后按下它时就会产生一个简单的叠层效果。`<TouchableHighlight>` 组件同时也为像 `onPressIn`、`onPressOut`、`onLongPress` 这样的事件提供了钩子，因此可以在 React 应用中使用这些事件。

例 4-3 展示了如何使用 `<TouchableHighlight>` 包装组件，以便为用户提供交互反馈。

例 4-3：使用 `<TouchableHighlight>` 组件

```
<TouchableHighlight
  onPressIn={this._onPressIn}
  onPressOut={this._onPressOut}
  style={styles.touchable}>
    <View style={styles.button}>
      <Text style={styles.welcome}>
        {this.state.pressing ? 'EEK!' : 'PUSH ME'}
      </Text>
    </View>
</TouchableHighlight>
```

当用户轻触按钮时，会产生叠层并改变文字（图 4-2）。

这是一个很勉强的例子，但它解释了一个基础的交互，即如何让移动应用中的按钮让人感觉是可触摸的。叠层是让用户感觉元素可触摸的一个关键的因素。注意，为了添加这样一个叠层，我们不需要为样式编写任何逻辑，`<TouchableHighlight>` 组件为我们处理了各种逻辑。

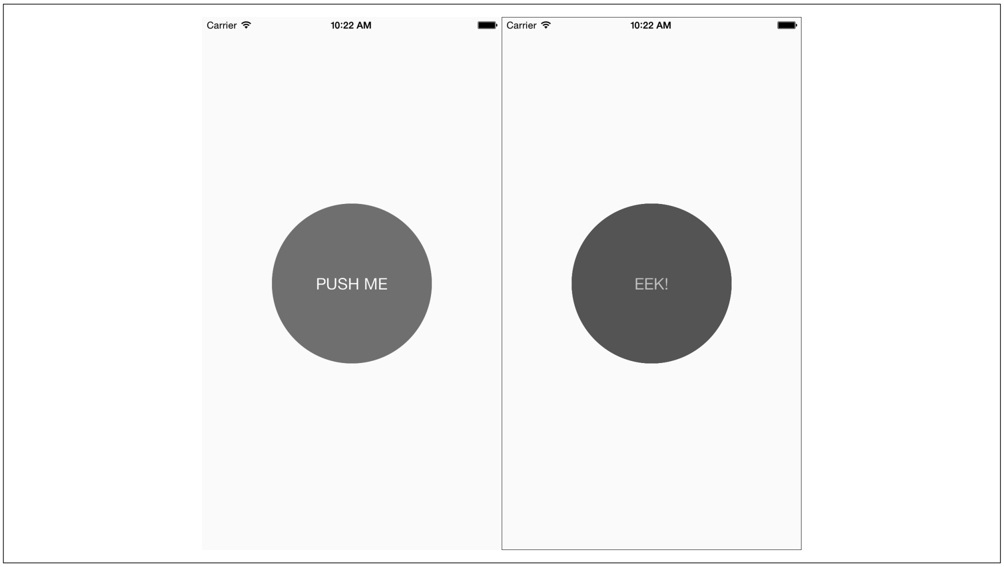


图 4-2：使用 <TouchableHighlight> 为用户提供视觉反馈——未按下状态（左）、按下状态（右）、带有高亮（右）

例 4-4 展示了这个按钮组件的完整代码。

例 4-4：Touch/PressDemo.js 说明了 <TouchableHighlight> 的用法

```
'use strict';

var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  TouchableHighlight
} = React;

var Button = React.createClass({
  getInitialState: function() {
    return {
      pressing: false
    }
  },

  _onPressIn: function() {
    this.setState({pressing: true});
  },

  _onPressOut: function() {
    this.setState({pressing: false});
  },
```

```
  render: function() {
    return (
      <View style={styles.container}>
        <TouchableHighlight
          onPressIn={this._onPressIn}
          onPressOut={this._onPressOut}
          style={styles.touchable}>

          <View style={styles.button}>
            <Text style={styles.welcome}>
              {this.state.pressing ? 'EEK!' : 'PUSH ME'}
            </Text>
          </View>

        </TouchableHighlight>
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
    color: '#FFFFFF'
  },
  touchable: {
    borderRadius: 100
  },
  button: {
    backgroundColor: '#FF0000',
    borderRadius: 100,
    height: 200,
    width: 200,
    justifyContent: 'center'
  },
});

module.exports = Button;
```

尝试编辑这个按钮，使其对其他事件作出响应，例如 `onPress` 或 `onLongPress`。想了解这些事件与用户交互的对应关系，最佳途径就是使用真实设备进行试验。

4.2.2 GestureResponder系统

如果你想要更多的交互，而不仅仅是轻触的话，React Native 也暴露了两个 API 用来处理

触摸逻辑：GestureResponder 和 PanResponder。GestureResponder 是一个低层的接口，而 PanResponder 提供了一些实用的抽象。我们首先将了解 GestureResponder（手势响应）系统的用法，因为它是 PanResponder 接口的基础。

移动设备上的触摸是相当复杂的，大多数的移动平台都支持多点触控，这意味着在同一时刻屏幕上会存在多个有效的触摸点（这些触摸点不一定都是手指，想象一下，也可能是用户的手掌碰到屏幕的边缘部分）。并且，如何确定哪个视图来处理触摸事件呢？这个问题与 Web 平台上鼠标事件的处理有些相似，默认的行为也很相似，即默认由最顶层的子节点来处理触摸事件。但是使用 React Native 的手势响应系统，如果需要的话，我们可以覆盖默认的行为。

触摸响应器（touch responder）是处理特定触摸事件的视图。前面我们看到 <Touchable-Highlight> 组件扮演了触摸响应器的角色。当然，我们也可以指定自己的组件为触摸响应器。协商过程的生命周期有一点复杂，如果一个视图想进入触摸响应器状态，则需要实现以下四个属性：

- View.props.onStartShouldSetResponder
- View.props.onMoveShouldSetResponder
- View.props.onResponderGrant
- View.props.onResponderReject

为了确定哪个视图进入响应状态，它们需要根据以下流程图（图 4-3）进行协调。

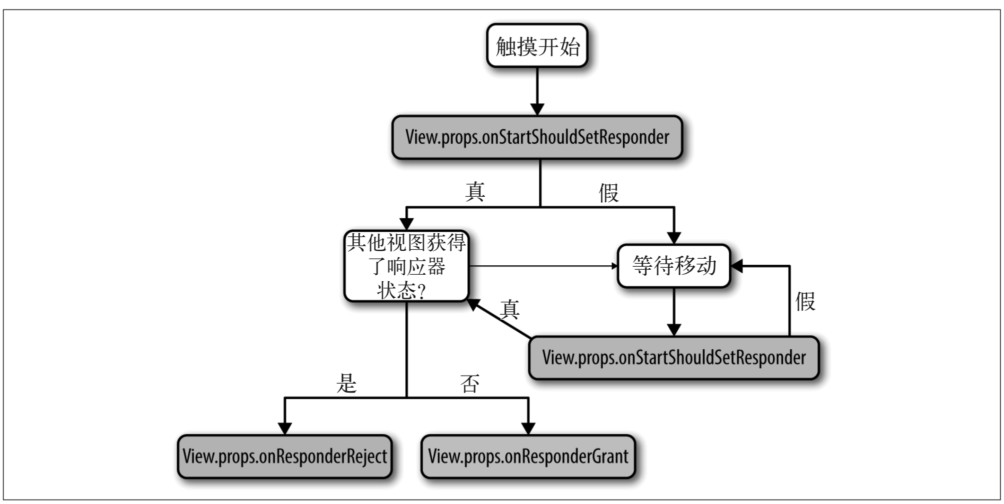


图 4-3：获取触摸响应器状态

呀，这看起来很复杂！让我们一起来梳理一下。首先，触摸事件有三个主要的生命周期阶段：开始、移动和释放（相当于浏览器中的 mouseDown、mouseMove 和 mouseUp）。一个视图

可以在开始或者移动阶段请求成为触摸响应器，这个行为通过 `onStartShouldSetResponder` 和 `onMoveShouldSetResponder` 来指定。当其中一个函数返回 `true`，这个视图就尝试申请进入响应状态。

当一个视图尝试申请响应状态之后，它可能被准许也可能被拒绝，相应的回调函数 `onResponderGrant` 或 `onResponderReject` 就会被调用。

响应器的协调函数采用冒泡的形式进行调用。也就是说，如果多个视图都尝试申请进入响应状态，那么嵌套最深的组件就会成为响应器。这通常是我们想要的行为，否则，你将很难在一个大视图内添加可触摸的按钮组件。如果你想覆盖这个行为，父组件可以使用 `onStartShouldSetResponderCapture` 和 `onMoveShouldSetResponderCapture` 函数，从而阻止子组件成为触摸响应器。

视图成功申请到响应状态之后，它相应的时间处理器将会被调用。以下是官方文档中手势响应器的摘要（http://facebook.github.io/react-native/docs/getting-started.html）。

- `View.props.onResponderMove`
  用户正在移动手指。
- `View.props.onResponderRelease`
  触摸结束被调用（也就是"touchUp"）。
- `View.props.onResponderTerminationRequest`
  其他元素想成为响应器。当前视图应该释放吗？返回 `true` 则允许释放。
- `View.props.onResponderTerminate`
  响应器被收回之后调用。它可能被其他视图通过调用 `onResponderTerminationRequest` 收回，或被操作系统强制收回（iOS 的控制中心或消息通知）。

大多数情况下，主要处理的是 `onResponderMove` 和 `onResponderRelease` 方法。

所有这些方法都接收一个合成的触摸事件对象，附带以下格式（同样摘录于文档）。

- `changedTouches`
  在上一次事件之后，所有发生变化的触摸事件的数组集合。
- `identifier`
  触摸点的标识符。
- `locationX`
  触摸点相对于父元素的横坐标。

- `locationY`

  触摸点相对于父元素的纵坐标。

- `pageX`

  触摸点相对于屏幕的横坐标。

- `pageY`

  触摸点相对于屏幕的纵坐标。

- `target`

  接受触摸事件的元素节点的标识符。

- `timestamp`

  触摸事件的时间戳，可用于移动速度的计算。

- `touches`

  当前屏幕上所有触摸点的集合。

当你需要决定是否响应触摸事件时，可以利用上面这些信息。例如，你的视图也许只关心两个手指的触摸。

这是相当低层的接口，如果你想使用这种方法进行手势的检测和响应，可能需要一些时间来调整正确的参数，并弄懂哪些值是你应该关注的。在下一节，我们将了解 `PanResponder`，它提供了一个用户手势更高层的抽象。

4.2.3 PanResponder

不像 `<TouchableHighlight>`，`PanResponder` 并不是一个组件，而是 React Native 的一个类，它提供了处理原生事件相对高层的接口。如官方文档（http://facebook.github.io/react-native/docs/getting-started.html）所述，`PanResponder gestureState` 对象为你提供了以下信息。

- `stateID`

  `gestureState` 的标识符（只要屏幕上至少有一个触摸点就会被保留）。

- `moveX`

  最近一次触摸移动时的屏幕横坐标。

- `moveY`

  最近一次触摸移动时的屏幕纵坐标。

- `x0`

  当响应器产生时的屏幕坐标。

- `y0`

  当响应器产生时的屏幕坐标。

- `dx`

  从触摸操作开始时的累计横向路程。

- `dy`

  从触摸操作开始时的累计纵向路程。

- `vx`

  当前的横向移动速度。

- `vy`

  当前的纵向移动速度。

- `numberActiveTouches`

  当前屏幕上的有效触摸点数量。

正如你所看到的，除原始位置的数据之外，`gestureState` 对象也包含了如当前触摸的速度和累计的距离等信息。

为了在组件中使用 `PanResponder`，我们需要创建一个 `PanResponder` 对象，然后将它附加到 `render` 方法中的组件上。

创建一个 `PanResponder` 需要我们为它指定对应的处理函数（例 4-5）。

例 4-5：创建 `PanResponder`，需要传入一些回调函数

```
this._panResponder = PanResponder.create({
  onStartShouldSetPanResponder: this._handleStartShouldSetPanResponder,
  onMoveShouldSetPanResponder: this._handleMoveShouldSetPanResponder,
  onPanResponderGrant: this._handlePanResponderGrant,
  onPanResponderMove: this._handlePanResponderMove,
  onPanResponderRelease: this._handlePanResponderEnd,
  onPanResponderTerminate: this._handlePanResponderEnd,
});
```

然后，使用传播语法（spread syntax）将 `PanResponder` 添加到 `render` 方法内的组件视图上（例 4-6）。

例 4-6：使用传播语法添加 `PanResponder`

```
render: function() {
  return (
    <View
      {...this._panResponder.panHandlers}>
      { /* 这里是视图内容 */ }
    </View>
```

```
    );
  }
```

之后，如果你的触摸起始于视图内，那么传入到 PanResponder.create 的处理函数将会在相应的移动事件发生时被调用。

例 4-7 展示了一个修改版的 PanResponder 的 React Native 示例代码。这个版本监听容器视图内的触摸事件，而不仅仅是圆形区域内。因此，当与应用互动时，你会看到数值被输出到屏幕上。如果你打算自己实现手势识别功能，建议你在真实设备上试验这个应用，这样你就可以知道这些值是怎样响应的。图 4-4 展示了这个例子的截图，但建议你在有触屏的设备上感受一下。

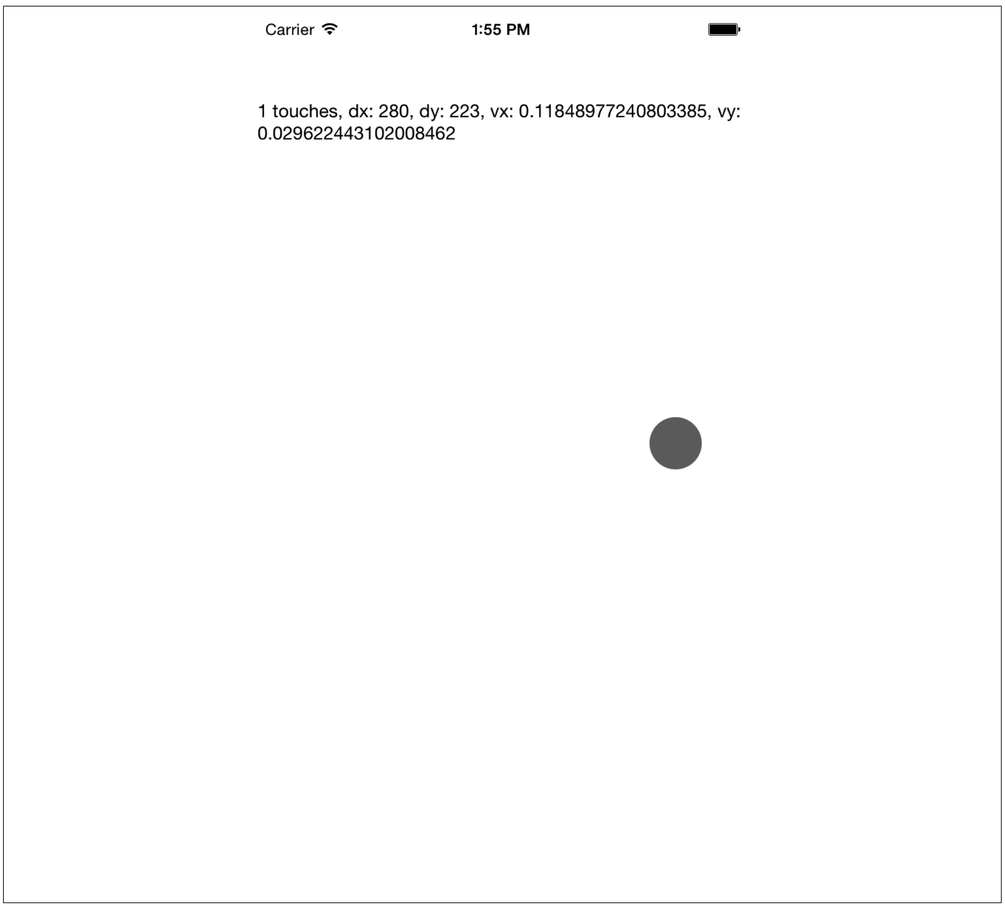


图 4-4：PanResponder 演示

例 4-7：Touch/PanDemo.js 说明了 PanResponder 的用法

```
// 改编自
// https://github.com/facebook/react-native/blob/master/
// Examples/UIExplorer/PanResponderExample.js

'use strict';

var React = require('react-native');
var {
  StyleSheet,
  PanResponder,
  View,
  Text
} = React;

var CIRCLE_SIZE = 40;
var CIRCLE_COLOR = 'blue';
var CIRCLE_HIGHLIGHT_COLOR = 'green';

var PanResponderExample = React.createClass({

  // 设置一些初始值。
  _panResponder: {},
  _previousLeft: 0,
  _previousTop: 0,
  _circleStyles: {},
  circle: null,

  getInitialState: function() {
    return {
      numberActiveTouches: 0,
      moveX: 0,
      moveY: 0,
      x0: 0,
      y0: 0,
      dx: 0,
      dy: 0,
      vx: 0,
      vy: 0,
    }
  },

  componentWillMount: function() {
    this._panResponder = PanResponder.create({
      onStartShouldSetPanResponder: this._handleStartShouldSetPanResponder,
      onMoveShouldSetPanResponder: this._handleMoveShouldSetPanResponder,
      onPanResponderGrant: this._handlePanResponderGrant,
      onPanResponderMove: this._handlePanResponderMove,
      onPanResponderRelease: this._handlePanResponderEnd,
      onPanResponderTerminate: this._handlePanResponderEnd,
    });

      this._previousLeft = 20;
      this._previousTop = 84;
      this._circleStyles = {
```

```
      left: this._previousLeft,
      top: this._previousTop,
    };
  },

  componentDidMount: function() {
    this._updatePosition();
  },

  render: function() {
    return (
      <View style={styles.container}>
        <View
          ref={(circle) => {
            this.circle = circle;
          }}
          style={styles.circle}
          {...this._panResponder.panHandlers}/>
        <Text>
          {this.state.numberActiveTouches} touches,
          dx: {this.state.dx},
          dy: {this.state.dy},
          vx: {this.state.vx},
          vy: {this.state.vy}
        </Text>
      </View>
    );
  },

// _highlight和_unHighlight被PanResponder方法调用,
// 给用户提供视觉反馈。
_highlight: function() {
  this.circle && this.circle.setNativeProps({
    backgroundColor: CIRCLE_HIGHLIGHT_COLOR
  });
},

_unHighlight: function() {
  this.circle && this.circle.setNativeProps({
    backgroundColor: CIRCLE_COLOR
  });
},

// 我们使用setNativeProps直接控制圆形的位置。
_updatePosition: function() {
  this.circle && this.circle.setNativeProps(this._circleStyles);
},

_handleStartShouldSetPanResponder:
function(e: Object, gestureState: Object): boolean {
   // 当用户按下圆形时,应该被激活吗?
   return true;
},

_handleMoveShouldSetPanResponder:
```

```
  function(e: Object, gestureState: Object): boolean {
    // 当用户触摸并移动圆形时,需要被激活吗?
    return true;
  },

  _handlePanResponderGrant: function(e: Object, gestureState: Object) {
    this._highlight();
  },

  _handlePanResponderMove: function(e: Object, gestureState: Object) {
    this.setState({
      stateID: gestureState.stateID,
      moveX: gestureState.moveX,
      moveY: gestureState.moveY,
      x0: gestureState.x0,
      y0: gestureState.y0,
      dx: gestureState.dx,
      dy: gestureState.dy,
      vx: gestureState.vx,
      vy: gestureState.vy,
      numberActiveTouches: gestureState.numberActiveTouches
    });

    // 使用差值计算当前位置。
    this._circleStyles.left = this._previousLeft + gestureState.dx;
    this._circleStyles.top = this._previousTop + gestureState.dy;
    this._updatePosition();
  },
  _handlePanResponderEnd: function(e: Object, gestureState: Object) {
    this._unHighlight();
    this._previousLeft += gestureState.dx;
    this._previousTop += gestureState.dy;
  },
});

var styles = StyleSheet.create({
  circle: {
    width: CIRCLE_SIZE,
    height: CIRCLE_SIZE,
    borderRadius: CIRCLE_SIZE / 2,
    backgroundColor: CIRCLE_COLOR,
    position: 'absolute',
    left: 0,
    top: 0,
  },
  container: {
    flex: 1,
    paddingTop: 64,
  },
});

module.exports = PanResponderExample;
```

选择处理触摸的方法

当使用上一节讨论的触摸和手势接口时，你应该如何选择呢？这取决于你想开发什么样的应用。

为了给用户提供基础的反馈，指明一个按钮或其他元素是可触摸的，你可以使用 `<TouchableHighlight>` 组件。

为了实现自己定制的触摸界面，你可以使用原始的手势响应系统或 `PanResponder`。大多数情况下，你应该更常用 `PanResponder` 方法，因为它也提供了基于手势响应系统的更简单的触摸事件。如果要设计一款游戏，或者一款有着与众不同的界面的应用，那就应该花一些时间使用这些接口开发你想要的交互方式。

对于许多应用来说，根本不需要使用手势响应系统或 `PanResponder` 来定制任何触摸处理函数。在下一节中，我们将接触一些更高级的组件，这些组件实现了通用的 UI 模式。

4.3 使用结构化组件

在这一节中，我们将了解结构化组件，并用它来控制应用的总体流向。结构化组件包括 `<TabView>`、`<NavigatorView>` 以及 `<ListView>` 等，它们都实现了一些移动应用中常见的交互和导航方式。一旦确定了应用导航的流向，你会发现这些组件在应用实现过程中会提供很大的帮助。

4.3.1 使用ListView

让我们从 `<ListView>` 组件的使用学起。在这一部分，我们将会开发一个《纽约时报》畅销图书列表的应用，并可以查看每一本图书的数据，如图 4-5 所示。如果你愿意的话，可以自己从《纽约时报》网站（http://developer.nytimes.com/apps/mykeys）抓取接口，或者也可以使用我们示例代码中的接口。

列表对于移动应用开发来说是极其实用的，并且你会发现许多应用的图形界面把列表当作中心元素来呈现。`<ListView>` 其实就是一系列视图的集合，通过它可以选择性地添加一些特殊视图，如分隔符、头部和尾部。可以参考 Dropbox、Twitter 和 iOS 设置的交互模式（图 4-6）。

`<ListView>` 是 React Native 发挥其长处的一个很好的例子，因为它可以很好地利用其宿主平台。在移动设备上，原生的 `<ListView>` 元素通常都是被高度优化的，因此渲染可以流畅无卡顿。如果你打算在 `<ListView>` 中渲染大量的列表项，最好保证子视图相对简洁，来尝试减少卡顿。

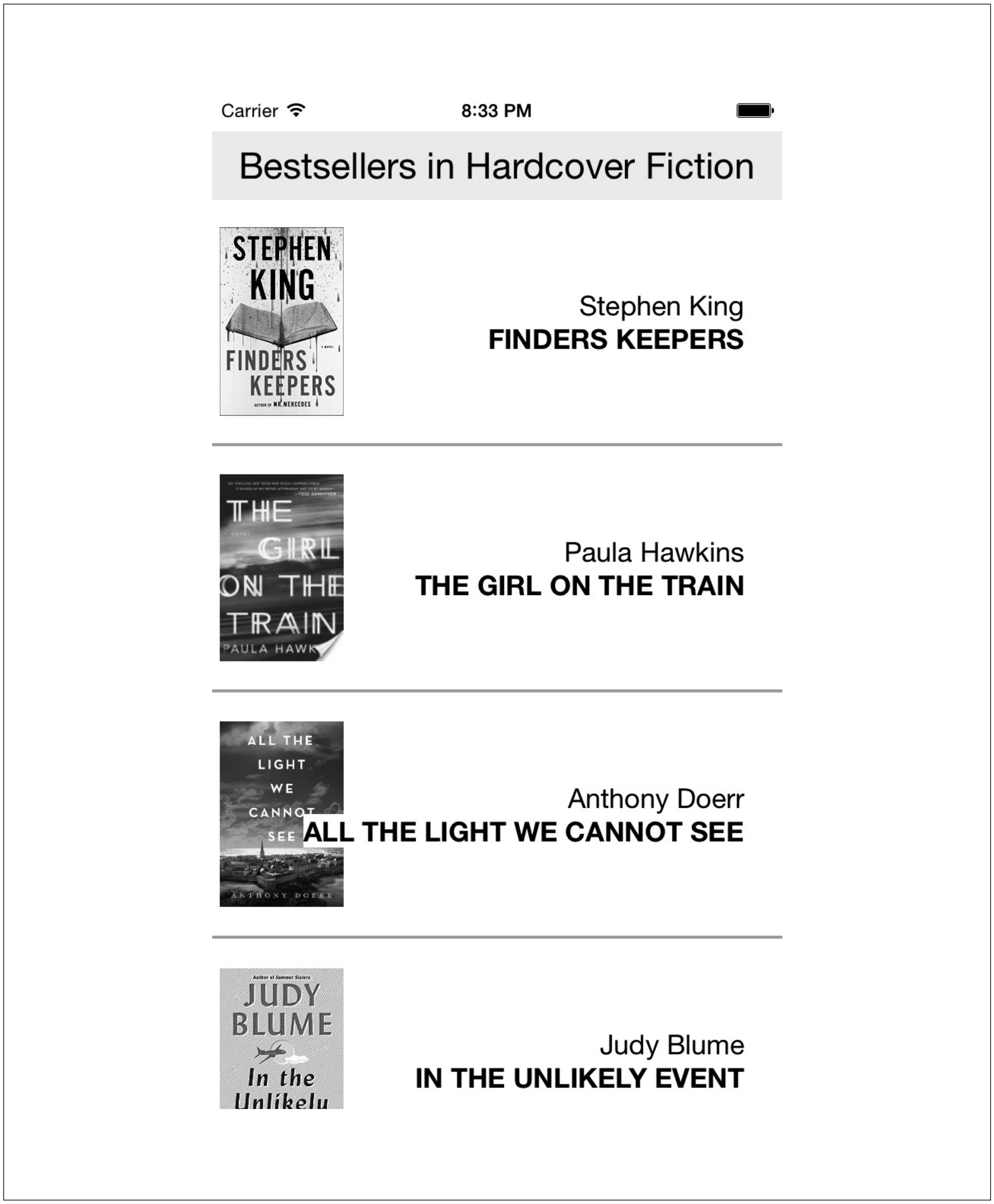


图 4-5：我们将要开发的图书列表应用

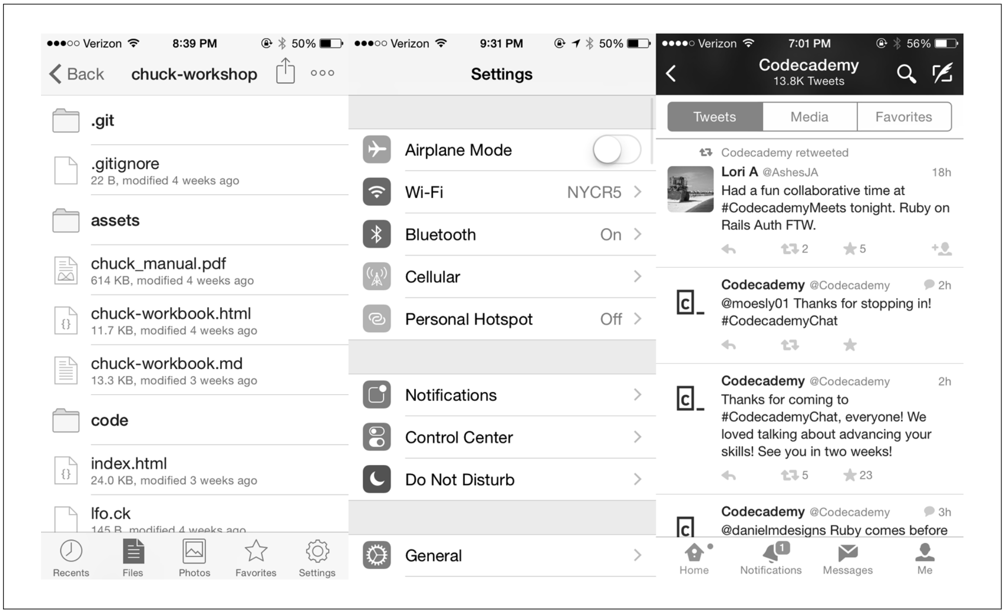


图 4-6：列表在 Dropbox、Twitter 和 iOS 设置中的使用

React Native 中的 `<ListView>` 组件至少需要两个属性：`dataSource` 和 `renderRow`。`dataSource` 顾名思义，是需要被渲染的源数据信息。`renderRow` 需要基于 `dataSource`（数据源）中的元素来返回一个组件。

基础的用法在 SimpleList.js. 文件中有说明。我们将从为 `<SimpleList>` 组件添加 `dataSource` 开始说起。一个列表的 `ListView.DataSource` 的数据源需要实现 `rowHasChanged` 方法。这里是一个简单的例子：

```
var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
```

为了设置 `dataSource` 的实际内容，我们使用 `cloneWithRows` 方法。让我们在 `getInitialState` 方法中返回 `dataSource`：

```
getInitialState: function() {
  var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
  return {
    dataSource: ds.cloneWithRows(['a', 'b', 'c', 'a longer example', 'd', 'e'])
  };
}
```

另一个需要的属性是 `renderRow`，它需要基于每一行的数据来返回一些 JSX：

```
_renderRow: function(rowData) {
  return <Text style={styles.row}>{rowData}</Text>;
}
```

现在可以将它们整合为一个简单的 <ListView>，通过渲染 <ListView> 为：

```
<ListView
  dataSource={this.state.dataSource}
  renderRow={this._renderRow}
  />
```

运行结果如图 4-7 所示。

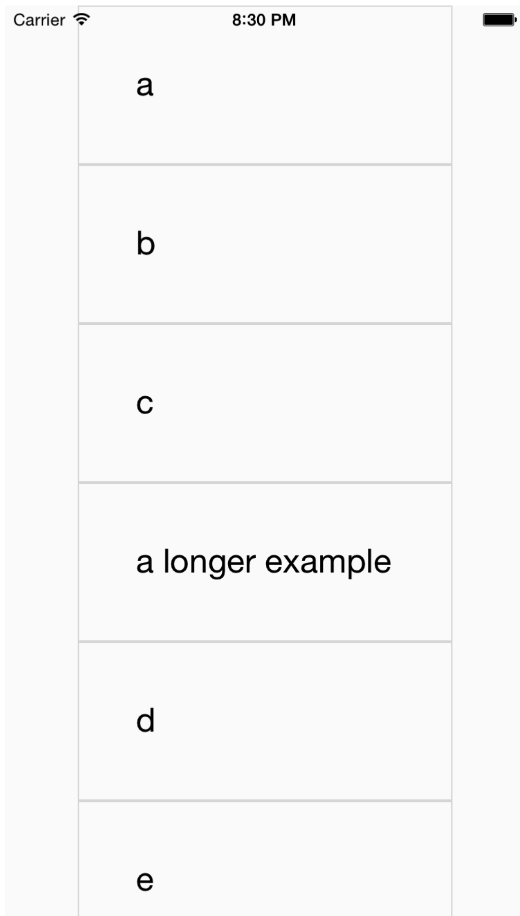


图 4-7：SimpleList 组件渲染了一个 <ListView> 的骨架

如果继续开发，要做些什么呢？我们来创建一个带有更复杂数据的 <ListView> 组件。我们使用纽约时报的接口来开发一个展现《纽约时报》畅销图书列表的简单应用。

首先，将数据初始化为空，因为要从远程获取数据：

```
getInitialState: function() {
  var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
  return {
    dataSource: ds.cloneWithRows([])
  };
}
```

紧接着，创建一个获取数据的方法，并在取得数据之后更新数据源。这个方法会在 componentDidMount 里被调用：

```
_refreshData: function() {
  var endpoint =
  'http://api.nytimes.com/svc/books/v3/lists/hardcover-fiction?response-format
  =json&api-key=' + API_KEY;
  fetch(endpoint)
    .then((response) => response.json())
    .then((rjson) => {
      this.setState({
        dataSource: this.state.dataSource.cloneWithRows(rjson.results.books)
      });
    });
}
```

每一本由纽约时报接口返回的图书数据都有三个属性：coverURL、author 和 title（封面图片地址、作者和标题）。我们更新 <ListView> 的 render 方法，让它基于这些属性返回组件（例 4-8）。

例 4-8：对于 _renderRow，我们只向 <BookItem> 传入相关的数据

```
_renderRow: function(rowData) {
  return <BookItem coverURL={rowData.book_image}
                   title={rowData.title}
                  author={rowData.author}/>;
},
```

我们再添加头部和尾部组件，例 4-9 解释了这部分是如何工作的。注意，<ListView> 的头部和尾部不是固定不动的，它们也会跟随列表一起滚动。如果你需要一个固定不动的头部和尾部，最简单的办法就是在 <ListView> 组件之外单独渲染。

例 4-9：为 BookListV2.js 中渲染头部和尾部添加相应的方法

```
_renderHeader: function() {
  return (<View style={styles.sectionDivider}>
    <Text style={styles.headingText}>
      Bestsellers in Hardcover Fiction
    </Text>
    </View>);
},

_renderFooter: function() {
  return(
    <View style={styles.sectionDivider}>
```

```
        <Text>
          Data from the New York Times Best Seller list.
        </Text>
      </View>
      );
  },
```

总体来说，畅销图书应用包括两个文件：BookListV2.js 和 BookItem.js，其中 BookListV2.js 如例 4-10 所示。（BookList.js 则是一个更简单的版本，省略了远程获取数据的功能，它也在 GitHub 仓库上，可以作为参考。）

例 4-10：Bestsellers/BookListV2.js

```
'use strict';

var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  Image,
  ListView,
} = React;

var BookItem = require('./BookItem');
var API_KEY = '73b19491b83909c7e07016f4bb4644f9:2:60667290';
var QUERY_TYPE = 'hardcover-fiction';
var API_STEM = 'http://api.nytimes.com/svc/books/v3/lists'
var ENDPOINT = `${API_STEM}/${QUERY_TYPE}?response-format=json&api-key=${API_KEY}`;

var BookList = React.createClass({
  getInitialState: function() {
    var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
    return {
      dataSource: ds.cloneWithRows([])
    };
  },

  componentDidMount: function() {
    this._refreshData();
  },

  _renderRow: function(rowData) {
    return <BookItem coverURL={rowData.book_image}
    title={rowData.title}
    author={rowData.author}/>;
  },

  _renderHeader: function() {
    return (<View style={styles.sectionDivider}>
      <Text style={styles.headingText}>
        Bestsellers in Hardcover Fiction
      </Text>
      </View>);
```

```
  },

  _renderFooter: function() {
    return(
      <View style={styles.sectionDivider}>
        <Text>Data from the New York Times Best Seller list.</Text>
      </View>);
  },

  _refreshData: function() {
    fetch(ENDPOINT)
      .then((response) => response.json())
      .then((rjson) => {
        this.setState({
          dataSource: this.state.dataSource.cloneWithRows(rjson.results.books)
        });
      });
  },

  render: function() {
    return (
        <ListView
          style=
          dataSource={this.state.dataSource}
          renderRow={this._renderRow}
          renderHeader={this._renderHeader}
          renderFooter={this._renderFooter}
          />
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#FFFFFF',
    paddingTop: 24
  },
  list: {
    flex: 1,
    flexDirection: 'row'
  },
  listContent: {
    flex: 1,
    flexDirection: 'column'
  },
  row: {
    flex: 1,
    fontSize: 24,
    padding: 42,
    borderWidth: 1,
    borderColor: '#DDDDDD'
  },
```

```
  sectionDivider: {
    padding: 8,
    backgroundColor: '#EEEEEE',
    alignItems: 'center'
  },
  headingText: {
    flex: 1,
    fontSize: 24,
    alignSelf: 'center'
  }
});

module.exports = BookList;
```

<BookItem> 是一个简单的组件，用来渲染列表中的每一个子视图（例 4-11）。

例 4-11：Bestsellers/BookItem.js

```
'use strict';

var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  Image,
  ListView,
} = React;


var styles = StyleSheet.create({
  bookItem: {
    flex: 1,
    flexDirection: 'row',
    backgroundColor: '#FFFFFF',
    borderBottomColor: '#AAAAAA',
    borderBottomWidth: 2,
    padding: 5
  },
  cover: {
    flex: 1,
    height: 150,
    resizeMode: 'contain'
  },
  info: {
    flex: 3,
    alignItems: 'flex-end',
    flexDirection: 'column',
    alignSelf: 'center',
    padding: 20
  },
  author: {
    fontSize: 18
  },
  title: {
```

```
      fontSize: 18,
      fontWeight: 'bold'
    }
  });

  var BookItem = React.createClass({
    propTypes: {
      coverURL: React.PropTypes.string.isRequired,
      author: React.PropTypes.string.isRequired,
      title: React.PropTypes.string.isRequired
    },

    render: function() {
      return (
        <View style={styles.bookItem}>
          <Image style={styles.cover} source=/>
          <View style={styles.info}>
            <Text style={styles.author}>{this.props.author}</Text>
            <Text style={styles.title}>{this.props.title}</Text>
          </View>
        </View>
        );
    }
  });

  module.exports = BookItem;
```

如果要展现很复杂的数据，或者很长的列表，你可能需要关注由 <ListView> 提供的一些可选的更复杂的属性来对它进行性能优化。当然，对于大多数用户而言，以上的用法已经足够了。

4.3.2 使用Navigator

<ListView> 是一个整合多视图以提供便利交互方式的很好的例子。在此基础上更进一步，我们还可以使用像 <Navigator> 这样的组件来展示应用的多个屏幕的内容，就像网站的多个页面一样。

<Navigator> 是一个难以捉摸但非常重要的组件，并且在许多通用型应用里被广泛使用。例如，iOS 设置中心就是结合了 <Navigator> 和多个 <ListView> 组件的产物（图 4-8）。Dropbox 应用也使用了 Navigator。

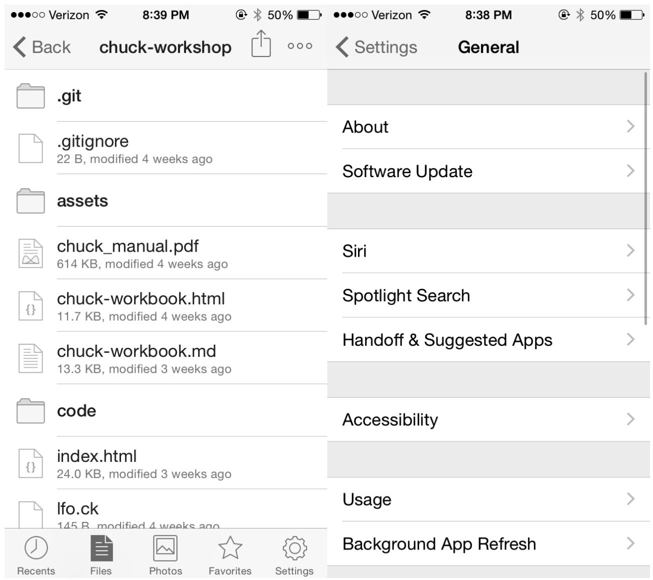


图 4-8：iOS 设置中心是 Navigator 行为的范例

<Navigator> 允许你的应用在不同屏幕之间切换（通常叫作“场景”）。它维护了一个“栈”路由，因此可以推入（push）、弹出（pop）和替换（replace）场景。你可以将其类比为浏览器的 history 接口。一个“路由”就是场景的名称和索引。

举例而言，在 iOS 设置中心，栈初始为空。当你选择一个子菜单之后，初始场景被推入栈中，点击左上角的“返回”按钮，场景就被弹出了。

如果你对这个组件感兴趣，UIExplorer 应用有很好的例子，教你使用各种 Navigator 的接口。

注意，实际上有两个 Navigator 可供选择：跨平台 <Navigator> 组件和 <NavigatorIOS> 组件。本书中，我们选择使用 <Navigator>。

我应该使用 NavigatorIOS 吗？

这真是个好问题！React Native 文档中有相同的问题（http://facebook.github.io/react-native/docs/navigator-comparison.html）。简单来说就是，你应该使用 <Navigator>。<NavigatorIOS> 不被核心团队支持，因此存在一些 bug。

具体来说，<Navigator> 是使用 JavaScript 重新实现的 Android 和 iOS 的 Navigator 组件。从这点上来说，它是完全跨平台且足够灵活的。iOS 平台的 <NavigatorIOS> 封装了 UIKit，因此获得了 Apple 的行为和动画，但是它的接口有一些限制，因为不是核心团队优先支持的，所以你应该不会想要使用它。

4.3.3　其他结构化组件

React Native 中还有许多结构化的组件。例如，实用的 <TabBarIOS> 和 <SegmentedControlIOS> 组件（图 4-9），以及 <DrawerLayoutAndroid> 和 <ToolbarAndroid> 组件等（图 4-10）。

你会发现这些组件都是以特定平台的名称为后缀命名的。这是因为它们为特定平台的 UI 元素封装了原生接口。

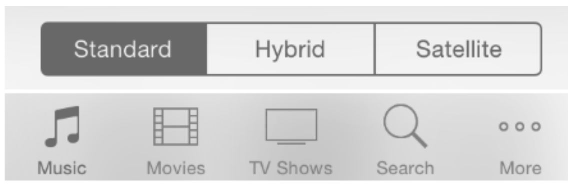


图 4-9：iOS 的 segmented 控制（上）和 tab 组件（下）

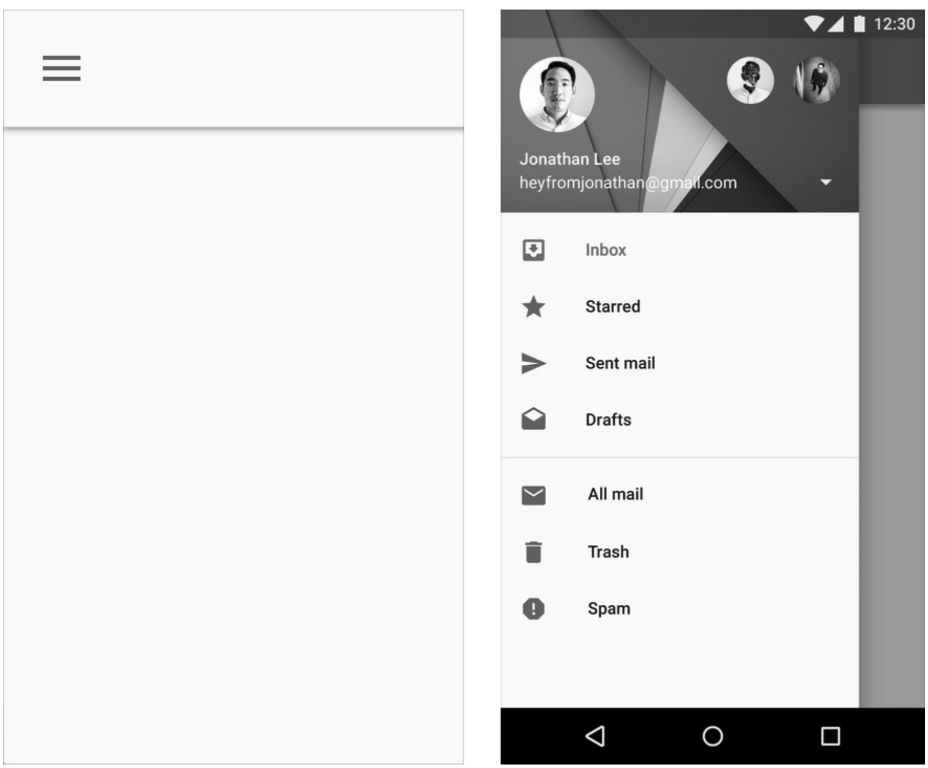


图 4-10：Android 的 toolbar（左）和 drawer 组件（右）

这些组件对于组织应用的多个界面来说是非常实用的。例如，<TabBarIOS> 和 <DrawerLayoutAndroid> 组件为你在模式和方法之间的切换提供了一种简便的方法。<SegmentedControlIOS> 和 <ToolbarAndroid> 适合用来做细粒度的控制。

推荐你参考特定平台的设计规范来更好地使用这些组件：

- Android 设计指南（https://developer.android.com/design/index.html）
- iOS 人机界面指南（https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/）

对了，等等！我们应该怎样使用平台特定的组件呢？我们先来看看怎样在跨平台应用中使用平台特定的组件。

4.4 平台特定组件

不是所有的组件都能在任何平台间通用的；同样，也不是所有的交互方式都适用于任何设备。但这并不意味着你不能在应用中使用平台特定的代码！在这一节中，我们将学习平台特定的组件，以及如何在跨平台应用上合并它们。

> 在 React Native 中编写跨平台代码并不是一个孤注一掷的做法！你可以在应用中混合跨平台代码和平台特定的代码，正如我们在这一节中的做法一样。

4.4.1 iOS或Android特定组件

有些组件只能在特定的平台上使用。包括 <TabBarIOS> 和 <SwitchAndroid> 在内的这些组件，它们通常是平台特有的，因为它们封装了一些平台特定的接口。对于一些组件来说，拥有平台无关的版本没有任何意义。例如，<ToolbarAndroid> 组件暴露了 Android 特有的接口，作用于一个 iOS 平台不支持的视图类型上。

平台特定的组件通常使用合适的后缀来命名：`IOS` 或 `Android`。如果你尝试在错误的平台引入组件，应用将会崩溃。

组件也可以拥有平台特定的属性，这些都在文档中有专门的标注，还有它们的用法。例如，<TextInput> 组件有一些平台无关的属性，也有一些限制只能在 iOS 或 Android 平台使用（图 4-11）。

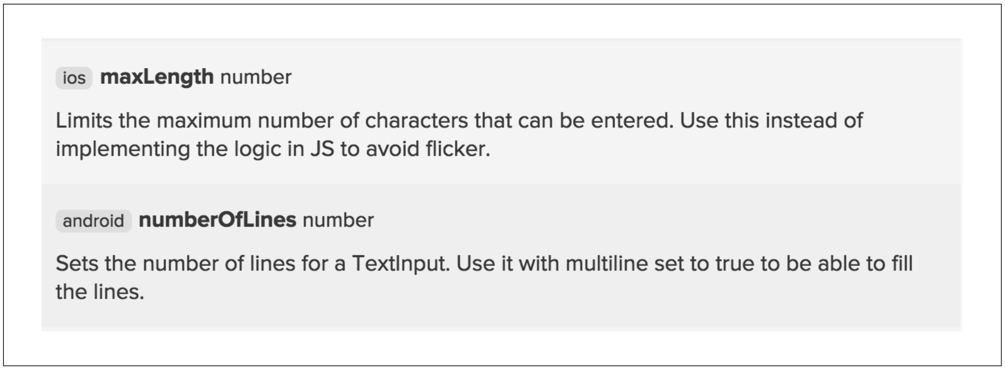


图 4-11：`<TextInput>` 有 Android 和 iOS 特定的属性

4.4.2　平台特定版本的组件

既然如此，应该怎样在跨平台应用中处理平台特定的组件或属性呢？好消息是，你可以继续使用这些组件。还记得我们有 index.ios.js 和 index.android.js 这两个文件吗？任何文件都可以使用这个命名规范针对 Android 或 iOS 进行不同的实现。

举个例子，我们可以使用 `<SwitchIOS>` 和 `<SwitchAndroid>` 组件。它们暴露了略微不同的接口，但如果只想要一个简单的切换组件呢？那么就可以创建一个包装组件 `<Swtich>`，让它根据平台渲染相应的组件。

我们将从实现 switch.ios.js（例 4-12）开始讲起。这是一个简单的 `<SwitchIOS>` 组件的包装，当切换组件被触发时会调用我们传入的回调函数。

例 4-12：Switch.ios.js

```
var React = require('react-native');
var { SwitchIOS } = React;

var Switch = React.createClass({
  getInitialState() {
    return {value: false};
  },

  _onValueChange(value) {
    this.setState({value: value});
    if (this.props.onValueChange) {
      this.props.onValueChange(value);
    }
  },

  render() {
    return (
      <SwitchIOS
        onValueChange={this._onValueChange}
```

```
        value={this.state.value}/>
      );
  }
});

module.exports = Switch;
```

接下来，我们一起来实现 switch.android.js（例 4-13）。

例 4-13：Switch.android.js

```
var React = require('react-native');
var { SwitchAndroid } = React;

var Switch = React.createClass({
  getInitialState() {
    return {value: false};
  },

  _onValueChange(value) {
    this.setState({value: value});
    if (this.props.onValueChange) {
      this.props.onValueChange(value);
    }
  },

  render() {
    return (
      <SwitchAndroid
        onValueChange={this._onValueChange}
        value={this.state.value}/>
      );
  }
});

module.exports = Switch;
```

我们注意到，它看起来跟 switch.ios.js 基本上是一样的，并且它也实现了相同的接口。唯一的区别在于它在内部使用了 `<SwitchAndroid>`，而不是 `<SwitchIOS>`。

现在可以用下面的语句从文件中导入 `<Switch>` 组件：

```
var Switch = require('./switch');
...
var switchComp = <Switch onValueChange={(val) => {console.log(val); }}/>;
```

接下来真正开始使用 `<Switch>` 组件。创建一个新的文件 CrossPlatform.js，其中包含例 4-14 中的代码。我们会实现一个基于 `<Switch>` 的数值来改变背景颜色的功能。

例 4-14：CrossPlatform.js 使用了 `<Switch>` 组件

```
var React = require('react-native');
var {
```

```
  StyleSheet,
  Text,
  View,
} = React;
var Switch = require('./switch');
var CrossPlatform = React.createClass({
  getInitialState() {
    return {val: false};
  },

  _onValueChange(val) {
    this.setState({val: val});
  },

  render: function() {
    var colorClass = this.state.val ? styles.blueContainer : styles.redContainer;
    return (
      <View style={[styles.container, colorClass]}>
        <Text style={styles.welcome}>
          Make me blue!
        </Text>
        <Switch onValueChange={this._onValueChange}/>
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  blueContainer: {
    backgroundColor: '#5555FF'
  },
  redContainer: {
    backgroundColor: '#FF5555'
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  }
});

module.exports = CrossPlatform;
```

你会发现其中 switch.js 文件并不存在，但是可以调用 `require(./switch)`。React 包管理器将会自动基于平台选择正确的实现，恰当地使用 switch.ios.js 或 switch.android.js 文件。

最后，替换 index.android.js 和 index.ios.js 文件的内容，让它渲染 `<CrossPlatform>` 这个组件（例 4-15）。

例 4-15：index.ios.js 和 index.android.js 文件应该对等，并简单地导入 crossplatform.js 文件

```
var React = require('react-native');
var { AppRegistry } = React;
var CrossPlatform = require('./crossplatform');

AppRegistry.registerComponent('PlatformSpecific', () => CrossPlatform);
```

现在，可以同时在 iOS 和 Android 平台运行应用了（图 4-12）。

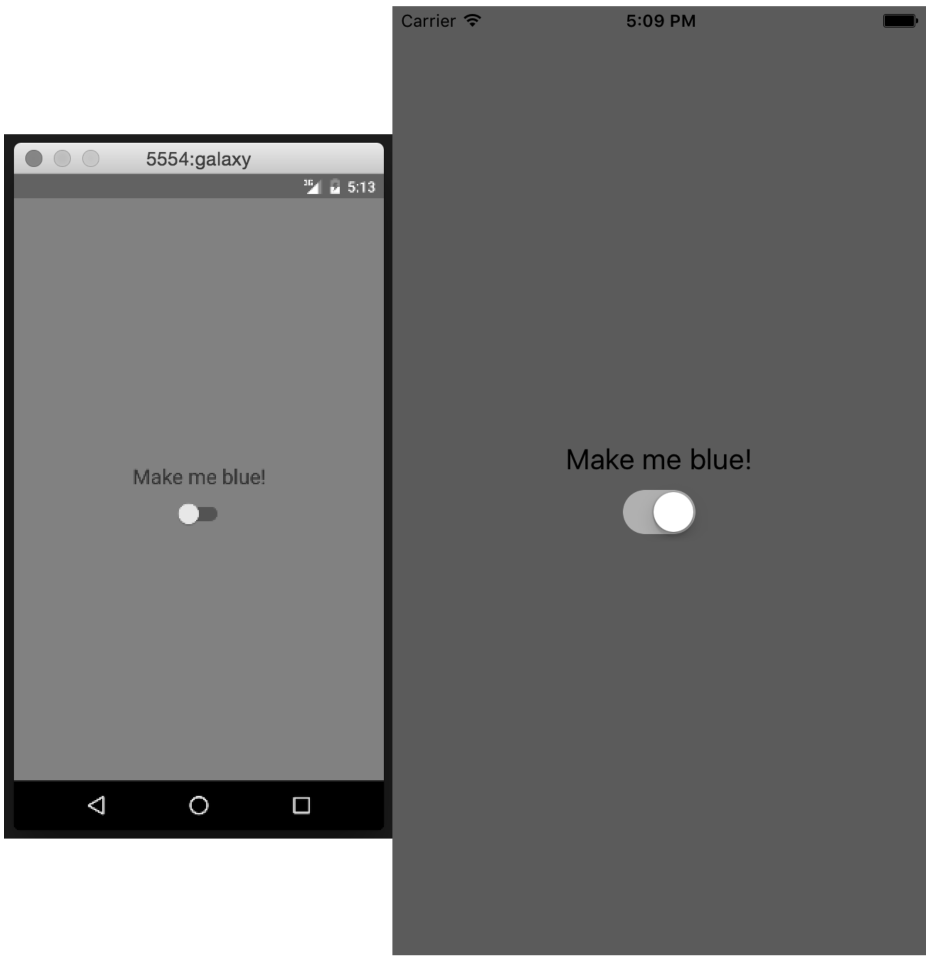


图 4-12：CrossPlatform 应用同时在 iOS 和 Android 上渲染相应的 <Switch> 组件

4.4.3 何时使用平台特定组件

那么，什么时候使用平台特定组件才合适呢？在多数情况下，当你的应用需要支持平台特定的交互方式的时候，你就应该使用它。当然，如果你希望自己的应用使用起来更“原生”的话，研究平台的界面规范是很有必要的。

Apple 和 Google 都为各自的平台提供了人机界面指南，值得一看：

- iOS 人机界面指南（https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/）
- Android 设计参考（https://developer.android.com/design/index.html）

通过创建平台特定版本的组件，可以更好地实现代码复用和平台定制之间的平衡。大部分情况下，为了同时支持 iOS 和 Android 平台，可能只需要用到少数分别支持特定平台的组件。

4.5 小结

在本章中，我们具体而深入地学习了 React Native 中最重要的一些组件。我们讨论了如何使用基础的低层组件，例如 `<Text>` 和 `<Image>`，以及像 `<ListView>`、`<Navigator>` 和 `<TabBarIOS>` 这样更高级的组件。同时，我们也学习了如何使用不同的触摸接口和组件，开发定制的触摸处理函数。最后，我们在实战中学习了使用平台特定组件的方法。

现在，你应该能使用 React Native 开发一些有基础功能的应用了！既然你已经了解了本章中讨论的组件，你会发现基于这些组件或结合这些组件来开发自己的应用与使用 Web 环境的 React 是非常相似的。

当然，能开发出具有基础功能的应用仅仅是我们的征程的一部分。在下一章中，我们将重点了解样式相关的内容，以及如何使用 React Native 的样式来达到想要的感观体验效果。

第 5 章

样式

能实现具体功能的应用固然是很不错的，但如果你不懂得如何为它添加样式，那么可能不会有很大的进展！在第 3 章中，我们使用基础样式构建了一个天气应用。虽然它让我们对 React Native 组件的样式有了一些大概的了解，但其中忽略了很多细节。本章中我们会更深入地学习 React Native 样式的用法，包括如何创建和管理样式表。当然，还有 React Native 实现 CSS 规则的细节。在学习之后，你应该可以轻松自如地为 React Native 组件或应用添加样式了。

如果想在 React Native 和 Web 应用之间共享样式，GitHub 上的这个 React Style 项目（https://github.com/js-next/react-style）提供了一种在 Web 上使用 React Native 样式的解决方案。

5.1　声明和操作样式

当使用 Web 环境的 React 时，我们通常使用分离的样式表文件，它们可能使用 CSS、SASS 或 LESS 编写。但 React Native 采用了一种完全不同的方式，它将样式完全带入了 JavaScript 的世界，强制你显式地链接样式和组件。毋庸置疑，这种方式引起了巨大的反响，因为它彻底摈弃了基于 CSS 的样式规范。

为了理解 React Native 样式的设计思想，首先需要考虑一些传统 CSS 样式表的痛点。[1] 传统 CSS 存在许多问题，所有的 CSS 规则和类名都在全局作用域里，如果不注意，一个组件样式很容易会影响到其他组件。例如，引入 Twitter 公司很流行的 Bootstrap 类库的同时也会

注 1：Christopher Chedeau, aka Vjeux 的“CSS in JS”幻灯片提供了一个很好的概览（https://speakerdeck.com/vjeux/react-css-in-js）。

引进 600 个新的全局变量。由于 CSS 并非显式地链接 HTML 元素，因此想消除无用的代码将会变得很困难，并且不容易确定哪种样式将会被应用到指定元素上。

像 SASS 和 LESS 这样的语言尝试替代 CSS 不尽如人意的部分，但许多类似的基本问题仍然存在。使用 React 让我们有机会保留 CSS 可取的部分，也可以避免一些有歧义的部分。React Native 实现了 CSS 可用样式的一个子集，在不损失高度表达能力的前提下能够保持样式接口的简洁性。然而，`position` 属性是完全不同的，我们在本章后续会学习到。并且，React Native 不支持伪类、动画或选择器。文档中可以查看支持的属性列表（https://facebook.github.io/react-native/docs/view.html#style）。

React Native 采用基于 JavaScript 的样式对象来代替传统样式表。它强制 JavaScript 代码和组件保持模块化，这也是 React Native 的巨大优势之一。通过把样式引入到 JavaScript 领域，React Native 让我们也可以编写模块化的样式。

在这一节中，我们将会学习如何在 React Native 中创建并操作样式对象。

5.1.1　内联样式

从语法上来看，内联样式是 React Native 中编写组件样式最简单的一种方法，虽然它们通常并不是最佳方式。正如例 5-1 所示，React Native 中内联样式的语法和浏览器上的 React 是一样的。

例 5-1：使用内联样式

```
<Text>
  The quick <Text style={{fontStyle: "italic"}}>brown</Text> fox
  jumped over the lazy <Text style={{fontWeight: "bold"}}>dog</Text>.
</Text>
```

内联样式有一些优势，它们简单粗暴，让你可以快速地调试。

但是由于它们比较低效，一般情况下应该避免使用。内联样式对象会在每一个渲染周期都被重新创建。即使你想根据 `props` 或 `state` 对样式作修改，也不一定需要使用内联样式，我们马上来看一看应该怎么做。

5.1.2　对象样式

如果你看过内联样式的语法，会发现可以给 `style` 属性传入一个对象。我们没有必要在每一次调用 `render` 方法时都重新创建样式对象，所以可以把它们分离出来，如例 5-2 所示。

例 5-2：`style` 属性接收一个 JavaScript 对象

```
var italic = {
  fontStyle: 'italic'
};
```

```
var bold = {
  fontWeight: 'bold'
};

...

render() {
  return (
    <Text>
      The quick <Text style={italic}>brown</Text> fox
      jumped over the lazy <Text style={bold}>dog</Text>.
    </Text>
    );
}
```

由 `Stylesheet.Create` 提供的不可变性很多时候弊大于利，例 4-7 的 PanDemo.js 就是一个很好的例子。回想一下，我们需要根据触摸运动更新圆形的位置；换言之，每一次我们从 `PanResponder` 接收到更新的数据，都需要去更新 `state` 以及圆形的样式。在这种情况下，根本不需要不可变性，至少控制圆形位置的样式是需要改变的。

因此可以使用简单对象来储存圆形的样式。

5.1.3　使用Stylesheet.Create

你会发现大多数的 React Native 示例代码都使用了 `Stylesheet.Create` 方法。`Stylesheet.Create` 方法是可选的，但一般都会使用它。文档（http://facebook.github.io/react-native/docs/style.html）是这么描述的：

> `Stylesheet.Create` 方法是可选的，但有一些重要的优势。它保证了值是不可变的，并且通过将它们转换成指向内部表的纯数字，保持了代码的不透明性。将它们放在文件的末尾可保证它们在应用中只会被创建一次，而不是每一次渲染周期都被重新创建。

换言之，`Stylesheet.Create` 其实只是提供保护措施的语法糖。绝大多数情况下，`Stylesheet.Create` 提供的不可变性是有益的。它也为你提供了通过 `propTypes` 校验属性的能力：通过 `Stylesheet.Create` 创建的样式可以通过 `View.propTypes.Style` 和 `Text.propTypes.Style` 类型进行校验。

5.1.4　样式拼接

如果想拼接两个以上的样式，应该怎么做呢？

先前已经提到过，相比于样式，我们更喜欢复用样式组件。这是正确的，但有时候可能也需要复用样式。例如，有一个按钮（`button`）样式和一个重点文本（`accentText`）的样式，你可能想结合二者来创建一个重点按钮（`AccentButton`）组件。

假设你的样式看起来是这样的：

```
var styles = Stylesheet.create({
  button: {
    borderRadius: '8px',
    backgroundColor: '#99CCFF'
  },
  accentText: {
    fontSize: 18,
    fontWeight: 'bold'
  }
});
```

然后通过简单地拼接样式创建了一个拥有两种样式的组件（例 5-3）。

例 5-3：style 属性也可以接收一个对象数组

```
var AccentButton = React.createClass({
  render: function() {
    return (
      <Text style={[styles.button, styles.accentText]}>
        {this.props.children}
      </Text>
    );
  }
});
```

正如你所看到的一样，style 属性可以接收一个样式对象数组。如果愿意的话，也可以在这里添加内联样式（例 5-4）。

例 5-4：可以混合样式对象与内联样式

```
var AccentButton = React.createClass({
  render: function() {
    return (
      <Text style={[styles.button, styles.accentText, {color: '#FFFFFF'}]}>
        {this.props.children}
      </Text>
    );
  }
});
```

如果遇到冲突，比如两个对象都指定了相同的属性，React Native 会帮我们解决冲突。样式数组中最右边的样式有最高优先权，并且空值（false、null 和 undefined）会被忽略。

你可以利用这个特性来使用条件性的样式，例如，我们有一个 <Button> 组件，希望它在被触摸的时候添加额外的样式，那么可以这样来编写代码（例 5-5）。

例 5-5：使用条件样式

```
<View style={[styles.button, this.state.touching && styles.highlight]} />
```

这种方式可以帮助保持渲染逻辑的简洁性。

总而言之，样式拼接是结合样式的一种实用的方法。对比一下 Web 样式的做法，在 SASS 中我们采用 `@extend` 关键字，或者在原生 CSS 中对类进行嵌套和重写。样式拼接是一种更为受限的工具，但它也有一定的长处：它保持了逻辑的简洁性，并且更容易推导出元素使用了何种样式以及如何使用的。

5.2 组织和继承

目前大多数的例子中都是将样式代码放在主 JavaScript 文件中，并通过调用 `Stylesheet.create` 来创建的。对于示例代码，这种方法是可行的，但并不适用于实际应用开发。那么应该怎样组织样式呢？在这一节中，我们将一起来学习组织样式的方法以及如何共享和继承样式。

5.2.1 导出样式对象

随着样式变得越来越复杂，你将会考虑把它们从组件 JavaScript 代码中分离出来。一种常用的方法是通过组件来划分目录。假设有一个名为 `<ComponentName>` 的组件，你可以创建一个名为 ComponentName/ 的目录，它的结构如下：

```
- ComponentName
  |- index.js
  |- styles.js
```

随后在 styles.js 文件中创建并导出样式表（例 5-6）。

例 5-6：从 JavaScript 文件中导出样式

```
'use strict';

var React = require('react-native');
var {
  StyleSheet,
} = React;

var styles = Stylesheet.create({
  text: {
    color: '#FF00FF',
    fontSize: 16
  },
  bold: {
    fontWeight: 'bold'
  }
});

module.exports = styles;
```

在 index.js 中，我们可以导入样式：

```
var styles = require('./styles.js');
```

接着，我们就可以在组件中使用了（例 5-7）。

例 5-7：从外部 JavaScript 文件中导入样式

```
'use strict';

var React = require('react-native');
var styles = require('./styles.js');
var {
  View,
  Text,
  StyleSheet
} = React;

var ComponentName = React.createClass({
  render: function() {
    return (
      <Text style={[styles.text, styles.bold]}>
        Hello, world
      </Text>
    );
  }
});
```

5.2.2 样式作为属性传递

你可以把样式作为属性进行传递。`View.propTypes.style` 这个属性类型确保只能传递有效的样式给属性。

你可以用这个方法开发出可扩展的组件，从而更有效地被父组件控制流程和操作样式。例如，一个组件接收一个可选的样式属性（例 5-8）。

例 5-8：组件通过属性接收样式对象

```
'use strict';

var React = require('react-native');
var {
  View,
  Text
} = React;

var CustomizableText = React.createClass({
  propTypes: {
    style: Text.propTypes.Style
  },
  getDefaultProps: function() {
    return {
      style: {}
    };
  },
```

```
    render: function() {
      return (
        <Text style={[myStyles.text, this.props.style]}>
          Hello, world
        </Text>
      );
    }
  });
```

通过把 `this.props.style` 放置在样式数组的末尾，我们确保可以重写默认属性。

5.2.3 复用和共享样式

通常我们更喜欢复用有样式的组件，而不是复用样式，但确实存在一些情况，使得我们需要在组件间共享样式。这时候，常用的办法是把项目大概组织成下面这样：

```
- js
  |- components
     |- Button
        |- index.js
        |- styles.js
  |- styles
     |- styles.js
     |- colors.js
     |- fonts.js
```

通过划分组件和样式到不同的目录中，你可以基于环境更清晰地保持每一个文件的预期用途。一个组件的目录应该包含 React 类，以及任何组件特定的文件。共享的样式应该放置在组件目录之外。共享样式可以包含像调色板、字体、标准内外边距等信息。

styles/styles.js 包含所有共享的样式文件，并进行了统一导出。然后你的组件可以导入 styles.js 文件，根据需要使用它们。或者，你可能更喜欢直接从 styles/ 目录导入特定的样式。

由于现在我们已经将样式移到了 JavaScript 中，怎样组织样式也成为了项目代码结构需要考虑的一部分，但是这里没有唯一正确的方法。

5.3 定位和设计布局

React Native 的样式功能使用中最大的一个改变是定位。CSS 支持多种定位技术，例如 `float`（浮动）、绝对定位、表格布局和块级布局等方法，这很容易使人迷惑。React Native 的定位方案则更加专注，主要依赖于 flexbox 和绝对定位，结合一些诸如 `margin` 和 `padding` 这样的属性。在这一节中，我们将学习如何在 React Native 中设计布局，并尝试开发一个蒙德里安画风（以抽象几何图案为特点）的布局应用。

5.3.1 使用flexbox布局

flexbox 是一个 CSS3 的布局模式。不像现有的块级和内联的布局方式，flexbox 给予我们一种无方向的设计布局的方案。（是的，垂直居中终于变得容易了！）React Native 重度依赖于 flexbox。如果你想阅读完整的说明，可以从 MDN 文档（https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Using_CSS_flexible_boxes）开始。

在 React Native 中，下列 flexbox 属性是可用的：

- `flex`
- `flexDirection`
- `flexWrap`
- `alignSelf`
- `alignItems`

并且，这些相关的属性也会影响布局：

- `height`
- `width`
- `margin`
- `border`
- `padding`

如果你曾经在 Web 平台使用过 flexbox，那么这里不会有太多的惊喜。flexbox 在 React Native 的设计布局中是至关重要的，所以这里我们将花一些时间来探索一下它的用法。

flexbox 背后主要的思想是创建可预见结构的布局，即使给定动态尺寸的元素也能够创建。由于我们为移动设备设计布局，需要适应多屏幕尺寸和方向，因此这是一个非常实用的功能。

我们从父元素 `<View>` 和一些子元素开始：

```
<View style={styles.parent}>
  <Text style={styles.child}> Child One </Text>
  <Text style={styles.child}> Child Two </Text>
  <Text style={styles.child}> Child Three </Text>
</View>
```

然后，为视图添加一些基础的样式，但尚未涉及定位：

```
var styles = StyleSheet.create({
  parent: {
    backgroundColor: '#F5FCFF',
    borderColor: '#0099AA',
    borderWidth: 5,
```

```
      marginTop: 30
    },
    child: {
      borderColor: '#AA0099',
      borderWidth: 2,
      textAlign: 'center',
      fontSize: 24,
     }
  });
```

最终的布局如图 5-1 所示。

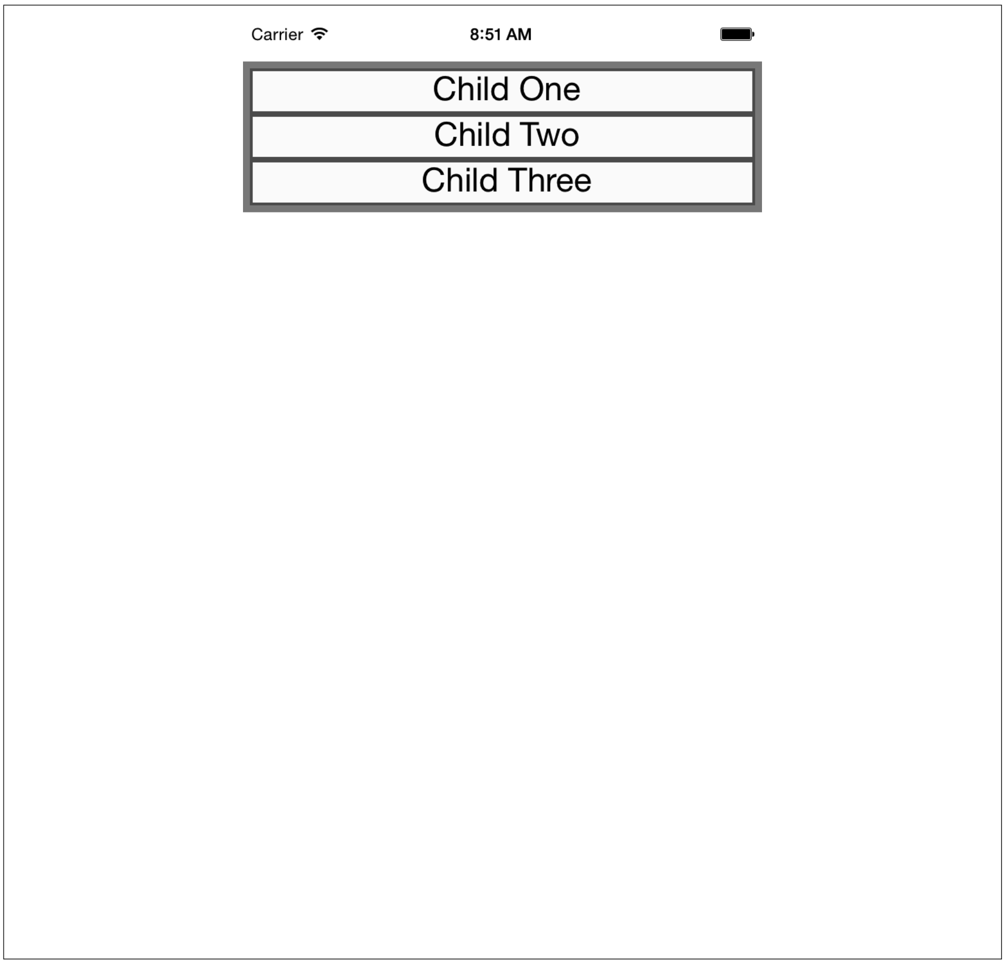


图 5-1：添加 flex 属性之前的布局情况

接下来，分别为父子元素添加 flex 属性。通过添加 flex 属性，我们明确地将它选为 flexbox 模式。flex 需要一个数字，这个数字决定了子元素获得相对权重的大小。把它们

都设置为 1，就获得了相等的权重。

设置 flexDirection: 'column'，使得组件纵向排列。如果设置 flexDirection: 'row'，那么子元素就会横向排列。改变的样式可以查看例 5-9。图 5-2 对比了不同值对布局的影响情况。

例 5-9：flex 和 flexDirection 属性的变化

```
var styles = StyleSheet.create({
  parent: {
    flex: 1,
    flexDirection: 'column',
    backgroundColor: '#F5FCFF',
    borderColor: '#0099AA',
    borderWidth: 5,
    marginTop: 30
  },
  child: {
    flex: 1,
    borderColor: '#AA0099',
    borderWidth: 2,
    textAlign: 'center',
    fontSize: 24,
  }
});
```

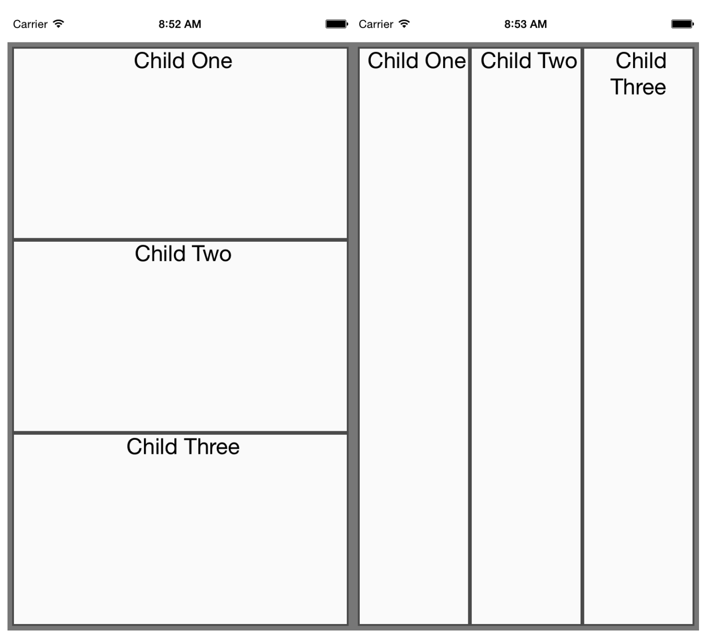


图 5-2：设置基础的 flex 和 flexDirection 属性；设置 flexDirection 属性为 column（左）和 row（右）

如果我们添加了 alignItems 属性，那么子元素将不再自动扩展填充水平和垂直两个方向上

可用的空间。我们设置了 flexDirection: 'row'，因此会自动填充行。然而现在它们只会尽可能占据垂直方向上它们所需的空间。

并且，alignItems 属性决定了它们在交叉轴上的位置。交叉轴与 flexDirection 正交，因此它们是相互垂直的。flex-start 会把子元素放置在顶部，center 将其居中，flex-end 则将其放置在底部。

让我们一起来看看添加 alignItems 属性之后发生了什么吧（结果如图 5-3 所示）。

```
var styles = StyleSheet.create({
  parent: {
    flex: 1,
    flexDirection: 'row',
    alignItems: 'flex-start',
    backgroundColor: '#F5FCFF',
    borderColor: '#0099AA',
    borderWidth: 5,
    marginTop: 30
  },
  child: {
    flex: 1,
    borderColor: '#AA0099',
    borderWidth: 2,
    textAlign: 'center',
    fontSize: 24,
  }
});
```

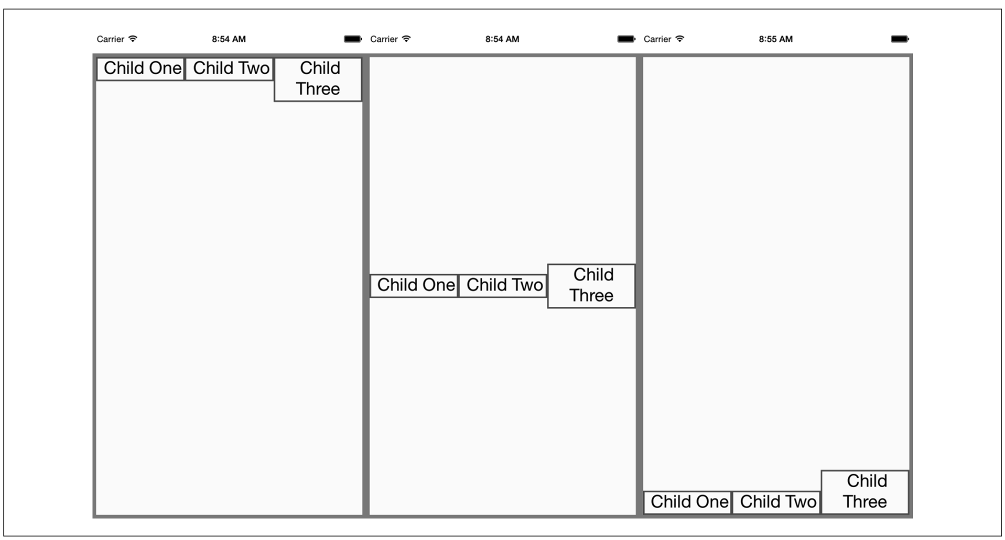


图 5-3：设置 alignItems 在交叉轴上定位子元素，交叉轴与 flexDirection 正交；这里分别设置了 flex-start、center 和 flex-end

5.3.2 使用绝对定位

除了 flexbox，React Native 也支持绝对定位，其工作方式基本上与 Web 上的一致，你可以通过设置 `position` 属性来启用绝对定位。

```
position: absolute
```

接着可以通过 `left`、`right`、`top` 和 `bottom` 这些熟悉的属性控制组件的定位。

一个绝对定位的子元素的坐标是相对于父元素的位置而存在的，因此你可以在父元素上使用 flexbox，然后在子元素上使用绝对定位。

但是此处仍然有一些限制，例如我们没有 `z-index` 属性，因此定位相互层叠视图会有一些复杂。一般来说，一组视图的最后一个元素有最高的优先级。

绝对定位非常实用。举例来说，如果你想在状态栏下创建一个容器视图，那么使用绝对定位就十分容易：

```
container: {
  position: 'absolute',
  top: 30,
  left: 0,
  right: 0,
  bottom: 0
}
```

5.3.3 学以致用

现在尝试使用定位技术来创建一个相对复杂的布局。先前提到过，我们要模仿蒙德里安的画风。图 5-4 是最终的布局效果。

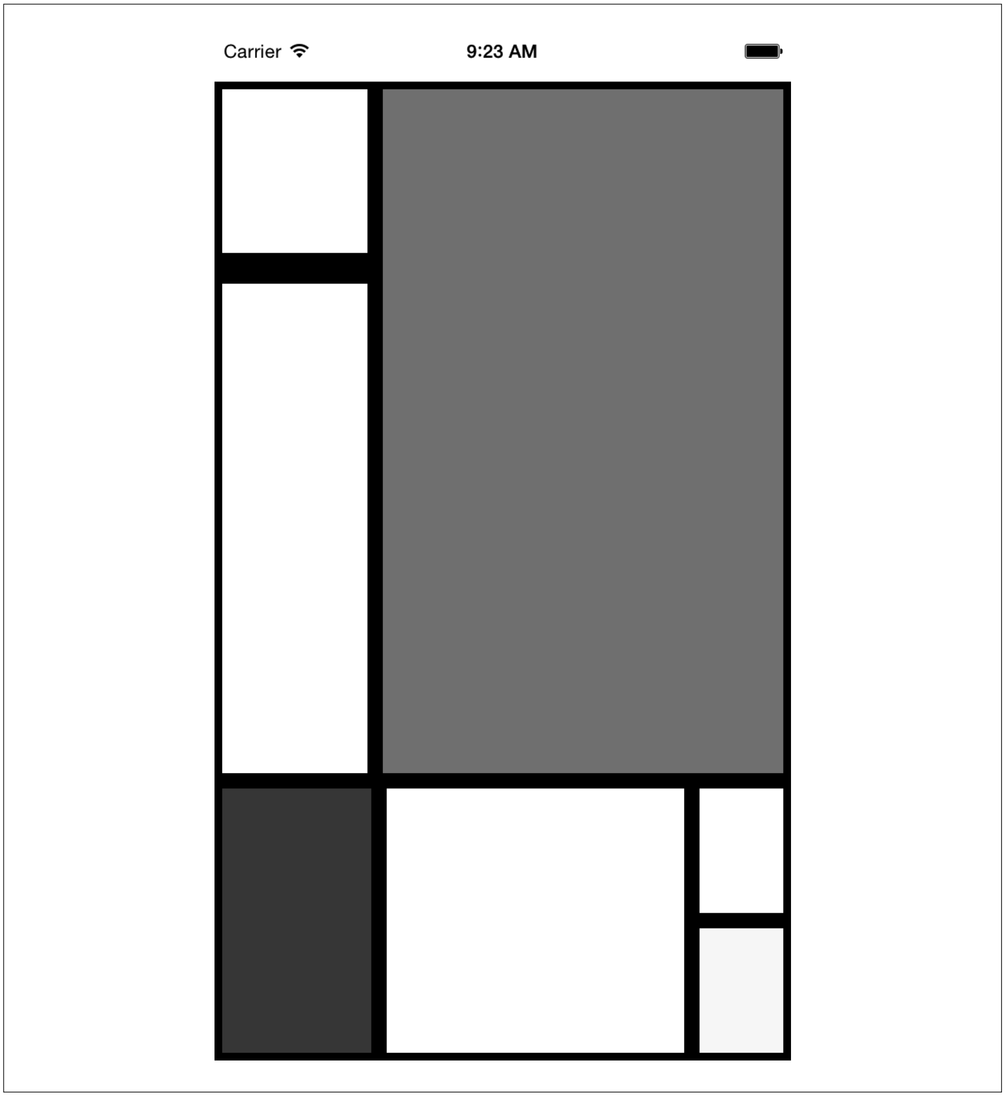


图 5-4：使用 flexbox 设计布局

我们应该怎样做才能实现这样的布局呢？

首先，创建一个 parent 样式作为容器。我们将会在父元素上使用绝对定位，因为这是最合适的：除了屏幕顶部状态栏的 30 像素的偏移之外，我们希望它可以填充所有的空间。我们同时也设置 flexDirection 为 column：

```
parent: {
  flexDirection: 'column',
  position: 'absolute',
```

```
    top: 30,
    left: 0,
    right: 0,
    bottom: 0
  }
```

再看看这张图，我们可以把它分割成较大的几个格子。这些分割都是任意的，因此选择任意一种方式切割即可。图 5-5 展示了分割布局的一种方式。

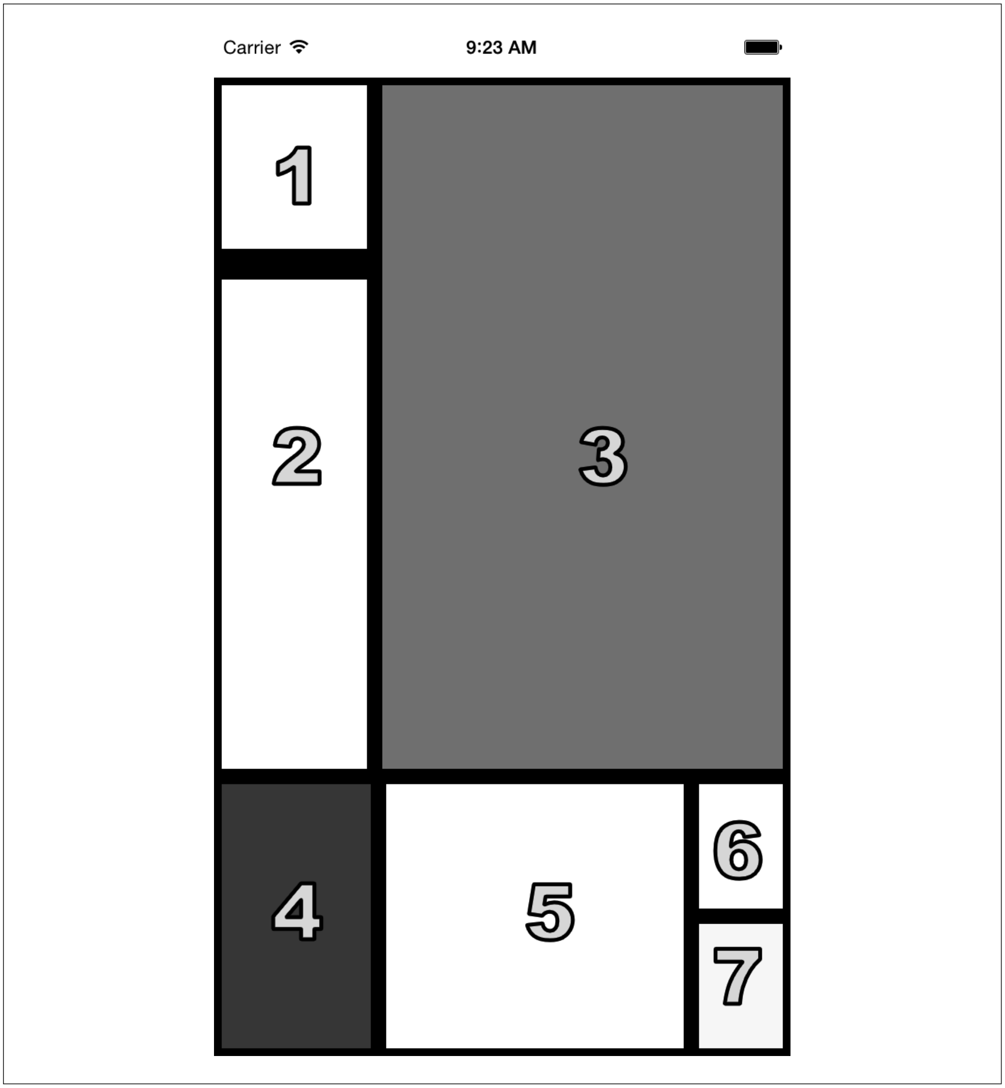


图 5-5：添加样式的顺序

起初我们把布局切分成顶部和底部两个格子：

```
<View style={styles.parent}>
  <View style={styles.topBlock}>
  </View>
  <View style={styles.bottomBlock}>
  </View>
</View>
```

然后创建下一层，它包含一个“左列”和“右底”部分，以及样式名为 cellThree、cellFour 和 cellFive 的 `<View>` 组件。

```
<View style={styles.parent}>
  <View style={styles.topBlock}>
    <View style={styles.leftCol}>
    </View>
    <View style={[styles.cellThree, styles.base]} />
  </View>
  <View style={styles.bottomBlock}>
    <View style={[styles.cellFour, styles.base]}/>
    <View style={[styles.cellFive, styles.base]}/>
    <View style={styles.bottomRight}>
    </View>
  </View>
</View>
```

最后一个部分包含了所有的七个 cell，例 5-10 展示了整个组件的代码。

例 5-10：Styles/Mondrian/index.js

```
'use strict';

var React = require('react-native');
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
} = React;
var styles = require('./style');

var Mondrian = React.createClass({
  render: function() {
    return (
      <View style={styles.parent}>
        <View style={styles.topBlock}>
          <View style={styles.leftCol}>
            <View style={[styles.cellOne, styles.base]} />
            <View style={[styles.base, styles.cellTwo]} />
          </View>
          <View style={[styles.cellThree, styles.base]} />
         </View>
         <View style={styles.bottomBlock}>
           <View style={[styles.cellFour, styles.base]}/>
           <View style={[styles.cellFive, styles.base]}/>
           <View style={styles.bottomRight}>
```

```
            <View style={[styles.cellSix, styles.base]} />
            <View style={[styles.cellSeven, styles.base]} />
          </View>
        </View>
      </View>
    );
  }
});

module.exports = Mondrian;
```

现在添加样式，使其生效（例 5-11）。

例 5-11：Styles/Mondrian/style.js

```
var React = require('react-native');
var { StyleSheet } = React;

var styles = StyleSheet.create({
  parent: {
    flexDirection: 'column',
    position: 'absolute',
    top: 30,
    left: 0,
    right: 0,
    bottom: 0
  },
  base: {
    borderColor: '#000000',
    borderWidth: 5
  },
  topBlock: {
    flexDirection: 'row',
    flex: 5
  },
  leftCol: {
    flex: 2
  },
  bottomBlock: {
    flex: 2,
    flexDirection: 'row'
  },
  bottomRight: {
    flexDirection: 'column',
    flex: 2
  },
  cellOne: {
    flex: 1,
    borderBottomWidth: 15
  },
  cellTwo: {
    flex: 3
  },
  cellThree: {
    backgroundColor: '#FF0000',
```

```
    flex: 5
  },
  cellFour: {
    flex: 3,
    backgroundColor: '#0000FF'
  },
  cellFive: {
    flex: 6
  },
  cellSix: {
    flex: 1
  },
  cellSeven: {
    flex: 1,
    backgroundColor: '#FFFF00'
  }
});

module.exports = styles;
```

5.4 小结

本章介绍了 React Native 样式的使用方法。虽然大部分原理与 Web 环境的 CSS 一致，但 React Native 为样式引入了一个不同的结构和使用方法。这里有许多新的内容需要消化！现在，你应该能够熟练使用 React Native 的样式功能来开发你想要的移动界面了。最妙的是，试验样式功能非常容易：模拟器上“重载”的功能赋予了我们紧凑的反馈回路。（在传统的移动应用开发中，编辑样式通常都需要重新构建应用，这很不值得。）

如果你想做更多样式方面的练习，可以回到前面的畅销图书应用或天气应用，尝试调整它们的样式和布局。后面的章节会开发更多示例应用，你也将得到更多素材用来练习。

第 6 章

平台接口

当你开发移动应用时，自然会想到使用宿主平台的特定接口。React Native 使开发者很容易就能使用诸如摄像头、地理定位和持久化存储这样的接口。React Native 通过引入模块的方式调用这些平台接口，并为我们提供了方便的异步 JavaScript 接口来调用底层的功能。

React Native 默认没有封装所有的宿主平台接口，一些平台接口需要你自己封装或者使用 React Native 社区封装的模块。第 7 章将会介绍这部分的内容。官方文档（https://facebook.github.io/react-native/docs/getting-started.html）是查看接口支持情况的最好的参考。

本章将介绍一些可用的平台接口。在本章的例子中，我们将对之前的天气应用作一些改动，为它添加地理定位的功能，使得应用可以自动检测用户的位置。此外我们还将添加“记忆”功能，使其能够保存先前的搜索记录。最后，我们允许用户自行从相册选择背景图片。

本章中每一节都会展示相应的代码片段；此外，该应用完整的代码包含在 6.4 节中。

iOS 与 Android 兼容性

这些平台接口的跨平台支持工作仍在进行中，例如 `AsyncStorage` 同时支持了 iOS 和 Android 平台，地理定位[1]和相册目前只支持 iOS 平台。查看已知问题列表（https://facebook.github.io/react-native/docs/known-issues.html）可以了解哪些模块正被移植到 Android 平台。

注 1：目前 Android 也支持了地理定位接口。——译者注

6.1 使用定位接口

对于移动应用来说，获取用户的定位信息是非常有用的。它允许你根据用户的相关信息更好地为用户提供服务。地理定位信息也在大量应用中被广泛使用。

值得庆幸的是，React Native 内置支持定位功能。这是一个平台无关的兼容性接口。它基于 MDN 的 Web 地理定位接口规范（https://developer.mozilla.org/en-US/docs/Web/API/Geolocation）返回数据。由于我们使用规范的定位接口，因此不需要操心平台相关的问题，比如地理服务以及编写完全兼容的位置感知的功能。

地理定位支持 iOS 与 Android 平台

目前 React Native 的地理定位接口已经同时支持了 iOS 和 Android 平台，详细信息可以查看官方文档。

6.1.1 获取用户地理位置

使用地理定位接口获取用户的位置信息轻而易举。如例 6-1 所示，我们需要调用 `navigator.geolocation`。

例 6-1：调用 `navigator.geolocation` 获取用户位置

```
navigator.geolocation.getCurrentPosition(
  (position) => {
    console.log(position);
  },
  (error) => {alert(error.message)},
  {enableHighAccuracy: true, timeout: 20000, maximumAge: 1000}
);
```

接口遵循了标准接口的规范，因此我们不需要单独导入它，非常容易使用。

`getCurrentPosition` 方法需要接收三个参数：成功回调函数、失败回调函数以及一系列的可选参数（`geoOptions`），其中成功回调函数是必需的。

传入成功回调函数的 `position` 对象包含了坐标信息和一个时间戳。例 6-2 展示了信息的格式和可能的取值。

例 6-2：调用 `getCurrentPosition` 的返回值格式样本

```
{
  coords: {
    speed:-1,
    longitude:-122.03031802,
    latitude:37.33259551999998,
    accuracy:500,
```

```
        heading:-1,
        altitude:0,
        altitudeAccuracy:-1
      },
      timestamp:459780747046.605
    }
```

geoOptions 必须是一个对象，它可以有这些键值：timeout、enableHighAccuracy 和 maximumAge。timeout 是其中最重要的参数，因为它可能会影响应用的逻辑。

6.1.2　处理权限问题

定位的数据属于敏感信息，因此默认情况下没有被启用。你的应用应该能够处理申请此权限被接受或被拒绝这两种情况。

大多数的移动应用平台都有定位权限这样的概念。比如在 iOS 平台上，用户可以选择完全阻止定位服务，或者逐一管理应用的权限。如果用户拒绝了应用的权限申请，那么 getCurrentPosition 中的失败回调函数将会被调用。

需要注意的是，定位权限本质上随时有可能被取消，因此建议应用程序做好定位服务调用失败的相关处理。

应用程序第一次申请定位服务时，用户将会看到如图 6-1 所示的权限对话框。

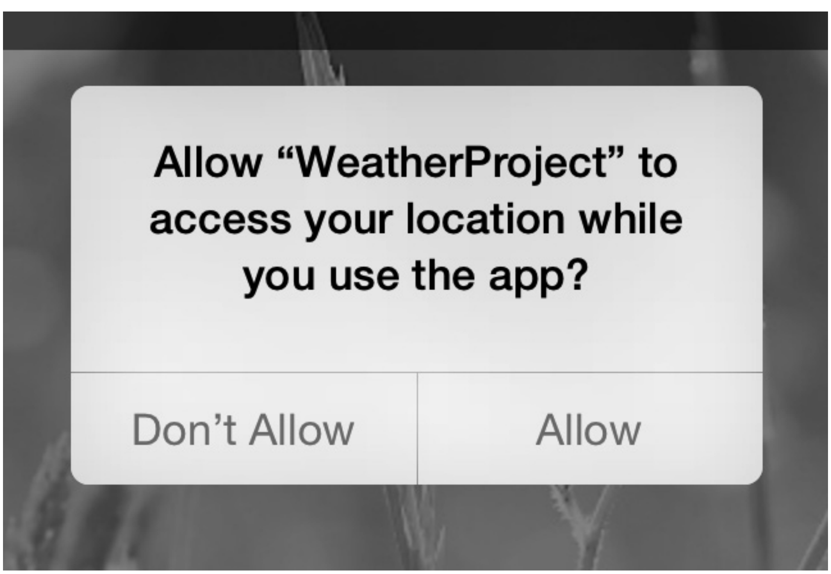


图 6-1：定位请求

当对话框处于激活状态时，不会触发回调函数。一旦用户点击任意一个选项之后，对应的回调函数就会被调用。设置会被保存起来，因此用户下一次不需要再选择。

假如用户拒绝了权限申请，如果愿意的话，你可以不作任何处理，但是大多数应用通常会重新申请一次权限。

6.1.3 在iOS模拟器上测试定位

你很可能会在模拟器上完成大部分的开发和测试工作，或者至少是在办公桌上完成的。那么怎样才能测试不同的位置呢？

iOS 模拟器可以轻而易举地模拟不同的位置，默认情况下会定位到美国加利福尼亚州附近的 Apple 公司总部，但是通过菜单中的“Debug → Location → Custom Location...”选项可以指定任何其他的坐标（图 6-2）。

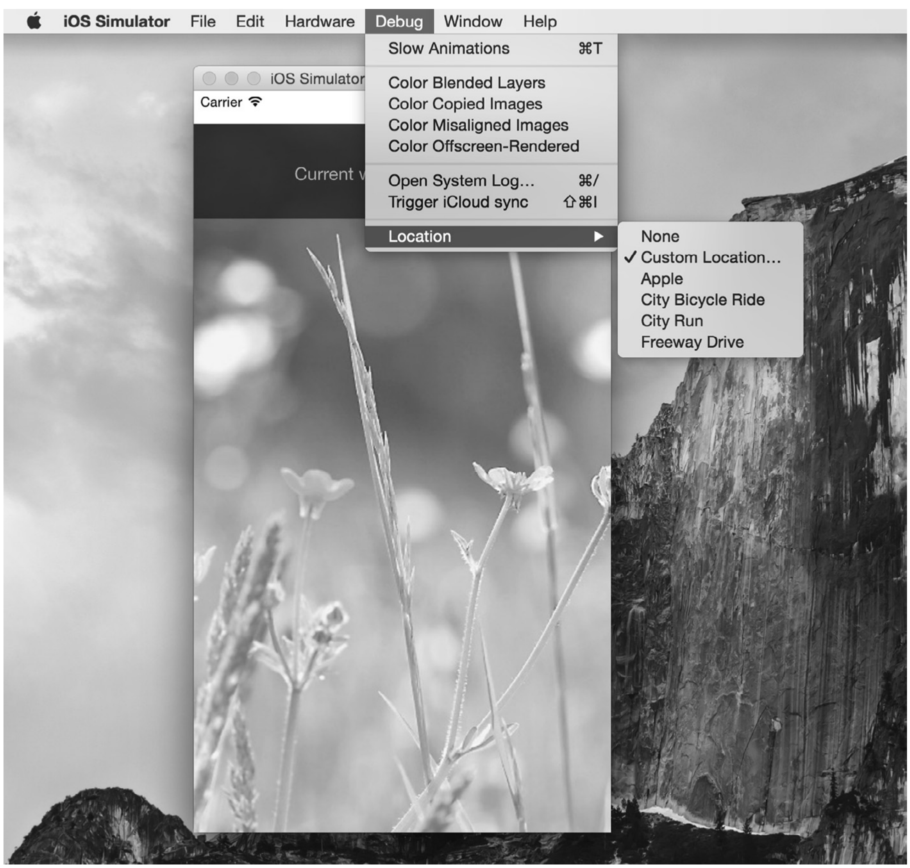


图 6-2：选择一个位置

通过选择不同的位置进行功能测试是个很好的实践方法，当然出于严谨考虑，你应该部署到真实物理设备上进行测试。

6.1.4　监听用户位置

你也可以监听用户的位置，每当位置变化时就会收到更新。这个功能可以用来长期记录用户的位置，或者用来保证应用接收到的是最新的位置信息：

```
this.watchID = navigator.geolocation.watchPosition((position) => {
  this.setState({position: position});
});
```

注意，你也可以在组件被卸载时清除监听器：

```
componentWillUnmount: function() {
  navigator.geolocation.clearWatch(this.watchID);
}
```

6.1.5　限制

由于定位接口基于 MDN 规范，因此它失去了一些更高级的基于定位的功能。例如：iOS 系统提供了“地理围栏”的接口，该接口会在用户进入指定地理区域（地理围栏）之后通知应用。React Native 暂时还不支持这个接口，这意味着如果你要使用基于定位的功能的话，需要自己移植接口。

6.1.6　改进天气应用

“智能天气应用”是之前天气应用的一个更新的版本，现在它利用了地理定位接口。你可以在图 6-3 中看到这些变化。

最值得一提的是 `<LocationButton>` 组件，它获取用户当前的位置并在点击之后触发回调函数。`<LocationButton>` 的代码在例 6-3 中。

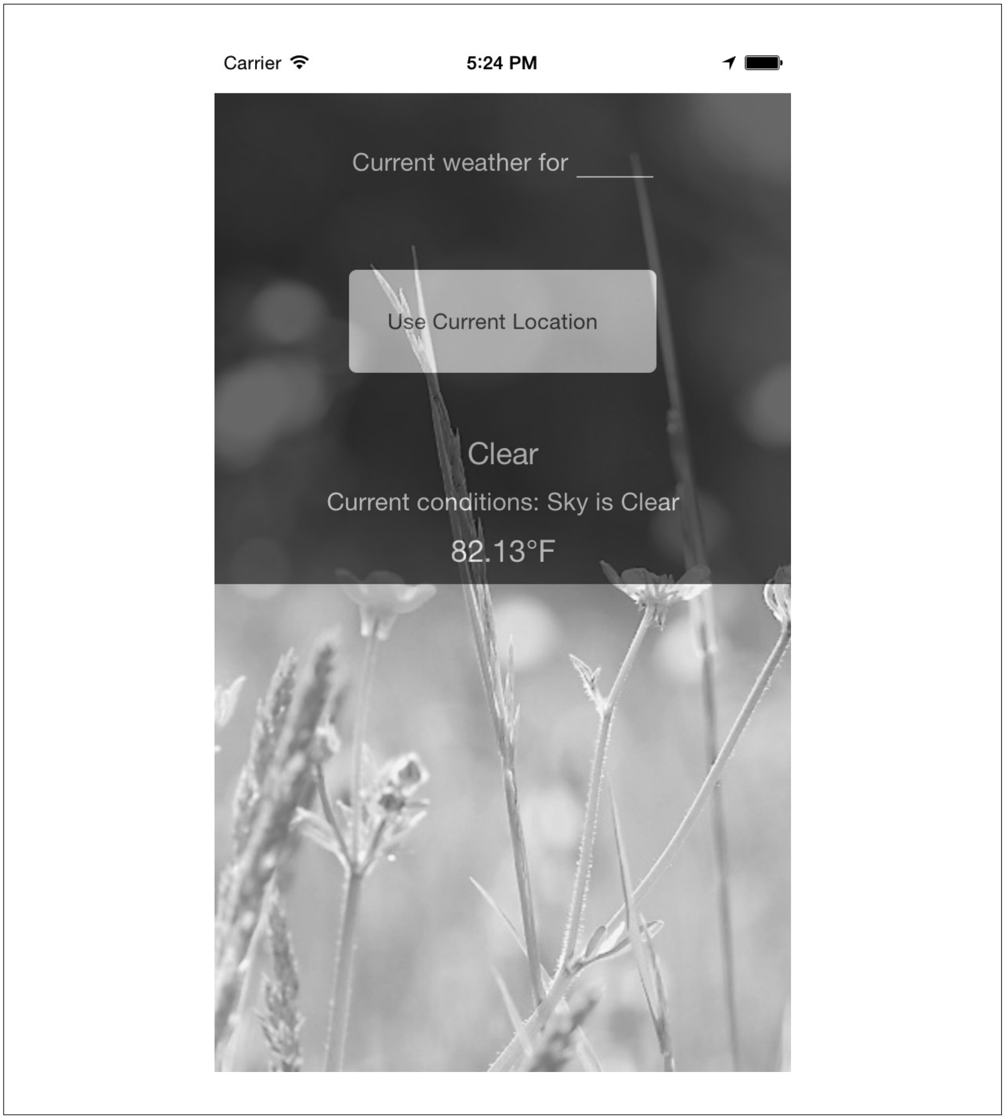


图 6-3：基于用户当前位置显示天气预报

例 6-3：SmarterWeather/LocationButton/index.js：点击按钮，获取用户位置

```
var React = require('react-native');
var styles = require('./style.js');
var Button = require('./../Button');

var LocationButton = React.createClass({
  propTypes: {
    onGetCoords: React.PropTypes.func.isRequired
  },
```

```
  _onPress: function() {
    navigator.geolocation.getCurrentPosition(
      (initialPosition) => {
        this.props.onGetCoords(initialPosition.coords.latitude,
          initialPosition.coords.longitude);
      },
      (error) => {alert(error.message)},
      {enableHighAccuracy: true, timeout: 20000, maximumAge: 1000}
    );
  },

  render: function() {
    return (
      <Button label="Use CurrentLocation"
        style={styles.locationButton}
        onPress={this._onPress}/>
      );
  }
});

module.exports = LocationButton;
```

LocationButton 使用的按钮组件代码见于本章结尾，它在 <TouchableHighlight> 中简单地封装了一个 <Text> 组件，并添加了一些基础的样式。

同时，我们也要更新 weather_project.js 主文件，使其支持两种查询方式（例 6-4）。幸运的是，OpenWeatherMap 接口同时支持基于经纬度和邮编的查询方式。

例 6-4：添加 _getForecastForCoords 和 _getForecastForZip 方法

```
var WEATHER_API_KEY = 'bbeb34ebf60ad50f7893e7440a1e2b0b';
var API_STEM = 'http://api.openweathermap.org/data/2.5/weather?';

...

_getForecastForZip: function(zip) {
  this._getForecast(
    `${API_STEM}q=${zip}&units=imperial&APPID=${WEATHER_API_KEY}`);
},

_getForecastForCoords: function(lat, lon) {
  this._getForecast(
    `${API_STEM}lat=${lat}&lon=${lon}&units=imperial&APPID=${WEATHER_API_KEY}`);
},

_getForecast: function(url, cb) {
  fetch(url)
    .then((response) => response.json())
    .then((responseJSON) => {
      console.log(responseJSON);
      this.setState({
        forecast: {
```

```
          main: responseJSON.weather[0].main,
          description: responseJSON.weather[0].description,
          temp: responseJSON.main.temp
        }
      });
    })
    .catch((error) => {
      console.warn(error);
    });
}
```

最后，我们在主视图导入 LocationButton 组件，将 _getForecastForCoords 作为回调函数。

```
<LocationButton onGetCoords={this._getForecastForCoords}/>
```

这里省略了样式的更新，因为应用完整的代码已经附在本章结尾。

如果你要正式上线给用户使用，还有大量的工作需要完成。例如，一个完整的应用应该有更好的错误提示以及更完善的界面反馈等。但是获取定位信息这样基础的功能确实是出乎意料地简单！

6.2　使用用户图片与摄像头

能够使用本地图片和摄像头是许多移动应用又一个重要的功能。在这一节中，我们将探索如何与用户图片数据和摄像头交互。

我们仍然使用“智能天气应用”这个项目，让它可以读取用户本地的图片作为应用背景。

6.2.1　相机模块

React Native 提供了一个相机的接口——获取用户设备本地的图片或从摄像头拍摄照片。



目前仅 iOS 支持相机模块

相机模块将很快支持 Android 平台，但是目前仅支持 iOS 平台。

与相机模块交互最基础的用法并不是太复杂。首先我们需要照常导入模块。

```
var React = require('react-native');
var { CameraRoll } = React;
```

然后，我们使用该模块来获取用户图片相关的信息，如例 6-5 所示。

例 6-5：CameraRoll.getPhotos 的基础用法

```
CameraRoll.getPhotos(
```

```
    {first: 1},
    (data) => {
      console.log(data);
    },
    (error) => {
      console.warn(error);
    });
```

我们通过合适的参数来调用 getPhotos 方法，它返回了一些相机图像的相关数据。

在“智能天气应用”中，我们用一个新的组件 PhotoBackdrop（例 6-6）替换最顶层的 <Image> 组件。目前 PhotoBackdrop 组件只是简单地从用户的相机获取最新图片。

例 6-6：SmarterWeather/PhotoBackdrop/camera_roll_example.js

```
var React = require('react-native');
var { Image, CameraRoll } = React;
var styles = require('./style.js');

var PhotoBackdrop = React.createClass({
  getInitialState() {
    return {
      photoSource: null
    }
  },
  componentDidMount() {
    CameraRoll.getPhotos(
      {first: 5},
      (data) => {
        this.setState({
          photoSource: {uri: data.edges[3].node.image.uri}
        })},
      (error) => {
        console.warn(error);
      });
  },
  render() {
    return (
      <Image
        style={styles.backdrop}
        source={ this.state.photoSource }
        resizeMode='cover'>
        {this.props.children}
      </Image>
      );
  }
});

module.exports = PhotoBackdrop;
```

CameraRoll.getPhotos 需要三个参数：一个带参数的对象、一个成功回调函数和一个错误回调函数。

6.2.2 通过getPhotoParams获取图片

`getPhotoParams` 对象接收一系列参数，不过奇怪的是这部分没有包含在文档中。可以从 React Native 源码（https://github.com/facebook/react-native/blob/master/Libraries/CameraRoll/CameraRoll.js#L46）查看哪些是可用的参数。

- `first`

  数字类型，想要逆序显示的照片数目（即最新保存的照片）。

- `after`

  字符串类型，一个匹配前一次调用 `getPhotos` 的 `page_info {end_cursor}` 信息的指针。

- `groupTypes`

  字符串类型，指定特定的组别来过滤结果。可能是 *Album*、*All* 和 *Event* 等值。完整的 `GroupTypes` 可在源码中查看。

- `groupName`

  字符串类型，在该组指定一个过滤器，例如 *Recent Photos* 或一个相册名称。

- `assetType`

  值为 *All*、*Photos* 或 *Videos* 的其中一个，为资源类型指定一个过滤器。

- `mimeTypes`

  字符串数组类型，基于 `MIME` 类型进行过滤（例如 image/jpeg）。

例 6-5 中简单使用了 `getPhotos` 方法，我们的 `getPhotoParams` 对象相当简洁：

```
{first: 1}
```

简单来说，这指明了我们要寻找最近的图片。

6.2.3 从相机渲染一张图片

从相机获取图片之后应该怎样渲染呢？让我们来看成功回调函数：

```
(data) => {
  this.setState({
    photoSource: {uri: data.edges[0].node.image.uri}
  })},
```

这个数据对象的结构不是非常直观，因此你可以使用调试器来审查对象。每一个 `data.edges` 中的对象都用一个节点（`node`）属性来表示一张图片；我们可以从这里获取实际资源的 URI。

你可能会想到，一个 <Image> 组件可以接收一个 URI 属性。所以，可以通过正确设置图片源属性的方式从相机渲染一张图片。

```
<Image source={this.state.photoSource} />
```

好了，现在可以看到包含图片的最终效果了，如图 6-4 所示。

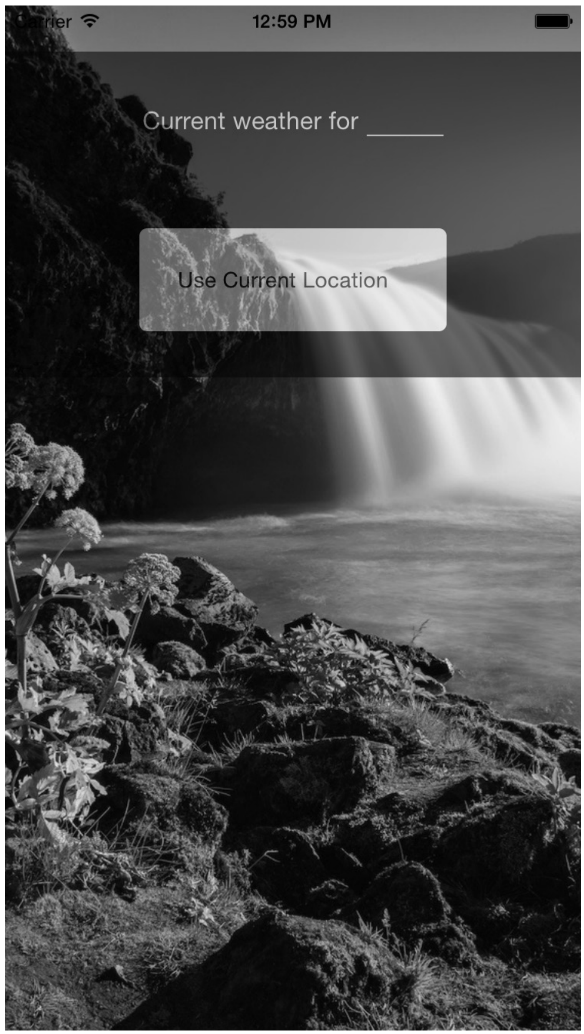


图 6-4：从相机渲染一张图片

6.2.4　展示照片列表

有很多应用为用户提供了选择图片的功能。那么我们应该怎样渲染图片选择的界面呢？

如果你是 iOS 用户，你可能发现 iOS 平台提供了默认的选择界面，但大多数应用实际上实现了它们自己定制的界面。如图 6-5 所示，Twitter 和 Tumblr 都有定制的界面。就 Twitter 而言，它允许你在发布界面选择图片。

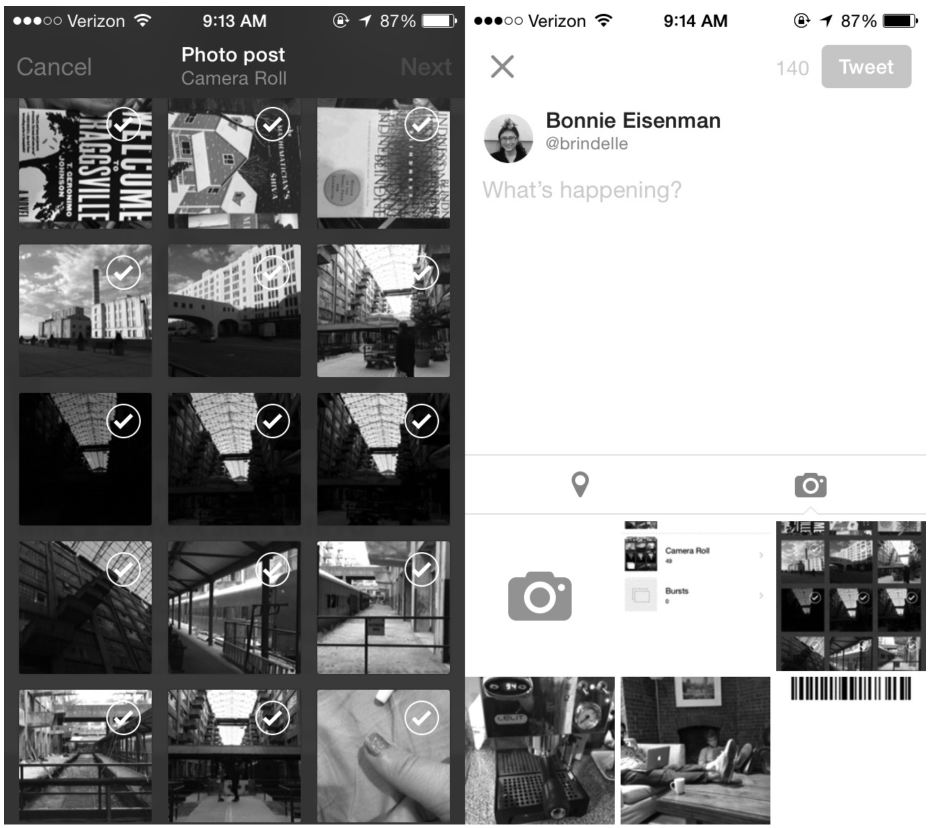


图 6-5：Tumblr（左）和 Twitter（右）iOS 应用中的图片选择界面

默认的图片选择界面是一个全屏的对话框，它看起来有点不同（图 6-6）。

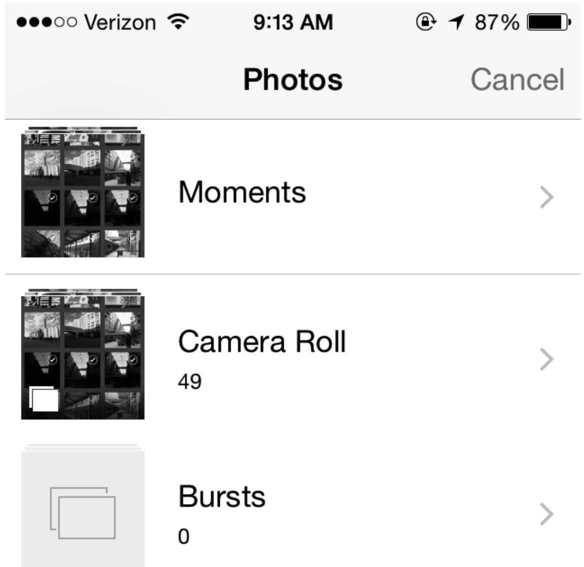


图 6-6：默认对话框

因此，你既可以使用内置的界面，也可以自己定制一个。应用通常为提供标准界面之外的功能而开发自己的定制方案。UIExplorer 应用（https://github.com/facebook/react-native/tree/master/Examples/UIExplorer）提供了一个非常基础的关于如何使用 `CameraRoll` 接口定制简易用户图片库的界面的例子，如图 6-7 所示。

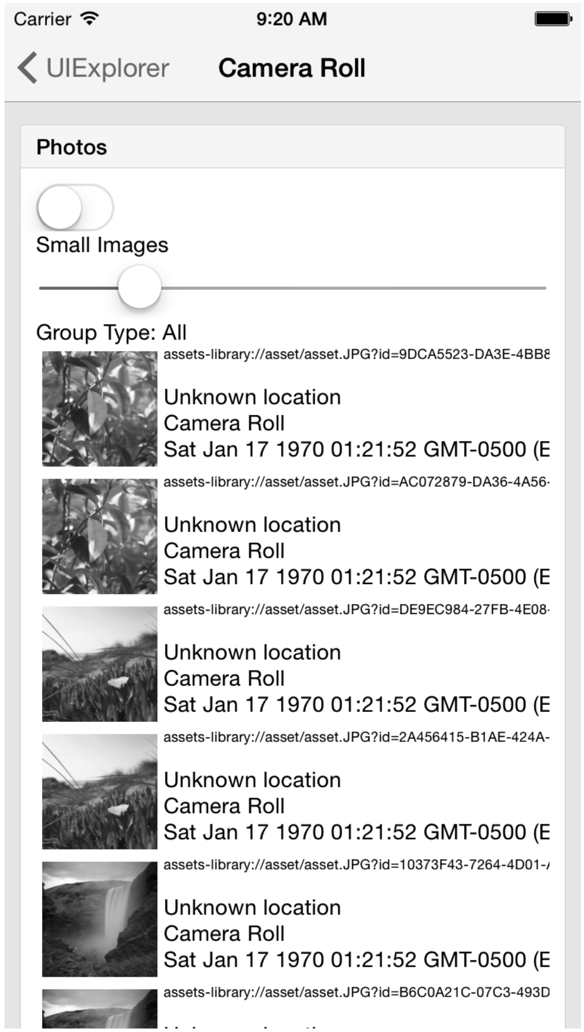


图 6-7：UIExplorer 应用的 `CameraRoll` 的例子

这是由先前介绍的相机交互的功能加上一个 `<ListView>` 组成的。不仅如此，还可以用这种方法开发一个同时支持 Android 和 iOS 的跨平台图片选择组件。

在 iOS 系统中，原生的界面元素是 `UIImagePickerController`，在 React Native 中通过 `ImagePickerIOS` 模块对它进行支持。

Android 照片选择的支持情况

目前，React Native 只为 iOS 提供了 `ImagePickerIOS` 接口用来选择照片或开启摄像头，但 Android 还未被支持。可以查看官文文档获取最新的信息（https://facebook.github.io/react-native/docs）。

可以通过下面这种通用的方式导入 `ImagePickerIOS` 模块。

```
var { ImagePickerIOS } = React;
```

接下来使用它就非常容易了，检测 `ImagePickerIOS` 以确认是否可以使用相机或进行视频录制（例 6-7）。

例 6-7：检测是否能通过 `ImagePickerIOS` 使用摄像头或录像

```
ImagePickerIOS.canUseCamera((result) => {
  console.log(result); // boolean
});

ImagePickerIOS.canRecordVideos((result) => {
  console.log(result); // boolean
});
```

然后，通过调用 `openSelectDialog` 触发一个照片选择的对话框，同时需要传入一些参数，比如成功选择照片之后的回调函数以及用户取消之后的回调函数（例 6-8）。

例 6-8：使用 `ImagePickerIOS` 触发照片选择对话框

```
ImagePickerIOS.openSelectDialog(
  {
    showImages: true,
    showVideos: false,
  },
  (data) => {
    this.setState({
      photoSource: {uri: data}
    });
  },
  () => {
    console.log('User canceled the action');
  });
```

这个调用会触发弹出一个标准的 iOS 图片选择对话框（图 6-8）。

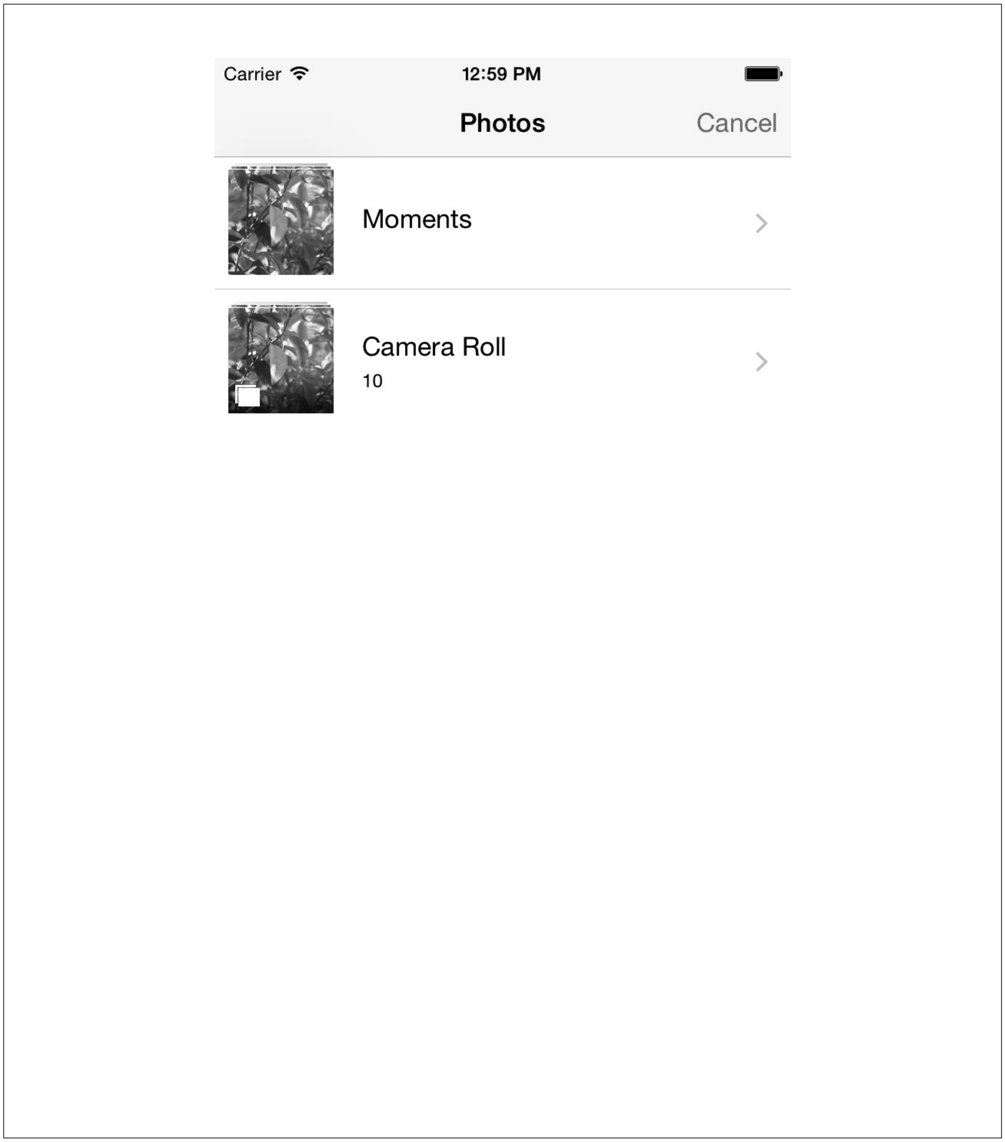


图 6-8：iOS 图片选择对话框

数据将会作为 URI 传入成功回调函数中，它可以在 <Image> 组件的源属性中使用。

6.2.5 上传图片至服务器

如果想上传照片到其他地方该怎么办呢？ React Native 的 XHR 模块包含了一个内置的图片上传功能，UIExplorer 应用讲解了一种方法（https://github.com/facebook/react-native/blob/0.16-stable/Examples/UIExplorer/XHRExample.ios.js）。

```
var formdata = new FormData();
...
formdata.append('image', {...this.state.randomPhoto, name: 'image.jpg'});
...
xhr.send(formdata);
```

XHR 是 XMLHttpRequest 的缩写。React Native 基于 iOS 网络接口实现了 XHR 接口。与定位接口类似，React Native 的 XHR 也是基于 MDN 规范（https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest）实现的。

相比于 Fetch 接口，使用 XHR 进行网络请求稍微有点复杂，但是基本用法就像例 6-9 这样。

例 6-9：通过 XHR 发送 POST 请求来上传照片的基本结构

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'http://posttestserver.com/post.php');
var formdata = new FormData();
formdata.append('image', {...this.state.photo, name: 'image.jpg'});
xhr.send(formdata);
```

这里省略了各种注册 XHR 请求的回调函数。

6.3 AsyncStore持久化数据存储

大多数的应用需要持续记录各式各样的数据。我们应该怎样在 React Native 中实现这个功能呢？

iOS 为我们提供了 AsyncStorage，一个应用于全局的键值对存储接口。如果你曾经使用过 Web 平台上的 LocalStorage，那么 AsyncStorage 应该会让你觉得非常熟悉。AsyncStorage 顾名思义，是一个异步的操作。这个接口也相当简洁，是一个 React Native 默认自带的模块。下面我们来看看如何使用它。

AsyncStorage 使用的储存键可以是任意字符串，它通常使用这样的格式：@AppName:key，例如：

```
var STORAGE_KEY = '@SmarterWeather:zip';
```

AsyncStorage 模块在调用 getItem 和 setItem 方法之后都会返回一个 Promise 对象。比如在“智能天气应用”中，我们可以在 componentDidMount 方法中加载存储的邮编：

```
AsyncStorage.getItem(STORAGE_KEY)
  .then((value) => {
    if (value !== null) {
      this._getForecastForZip(value);
    }
  })
  .catch((error) => console.log('AsyncStorage error: ' + error.message))
  .done();
```

接着，在 `_getForecaseForZip` 方法中，我们储存邮编：

```
AsyncStorage.setItem(STORAGE_KEY, zip)
  .then(() => console.log('Saved selection to disk: ' + zip))
  .catch((error) => console.log('AsyncStorage error: ' + error.message))
  .done();
```

`AsyncStorage` 同时也提供了删除键值、合并键值和获取所有可用键值等方法。

其他存储方式

如果你需要处理复杂的、结构化的数据，或者仅仅是更多的数据，你很可能不满足于简单的键值对存储。

有一个 iOS 通用的数据库 SQLite 可供使用，不过它不是 React Native 内置的模块。下一章将会介绍如何为 React Native 包装本地模块，以及如何安装别人编写的模块。

6.4 智能天气应用

本章中所有的例子都可以在 SmarterWeather/ 目录下找到。该应用在第 3 章的基础上加以修改，已经有了一些变化，因此我们再看一下整个应用的结构（图 6-9）。

```
SmarterWeather/
├── Forecast
│   └── index.js
├── LocationButton
│   ├── index.js
│   └── style.js
├── PhotoBackdrop
│   ├── camera_roll_example.js
│   ├── index.js
│   ├── local_image.js
│   └── style.js
├── android
│   ├── app
│   ├── build
│   ├── build.gradle
│   ├── gradle
│   ├── gradle.properties
│   ├── gradlew
│   ├── gradlew.bat
│   └── settings.gradle
├── index.android.js
├── index.ios.js
├── ios
│   ├── SmarterWeather
│   ├── SmarterWeather.xcodeproj
│   ├── SmarterWeatherTests
│   └── main.jsbundle
├── node_modules
│   └── react-native
├── package.json
├── styles
│   └── typography.js
└── weather_project.js
```

图 6-9：智能天气应用的项目内容

顶层组件位于 weather_project.js 中。共享字体样式位于 styles/typography.js 中。Forecast/、PhotoBackdrop/、Button/ 和 LocationButton/ 目录包含了智能天气应用中所有的 React 组件。

6.4.1 WeatherProject组件

顶层组件在 weather_project.js 文件里（例 6-10）。这里包含了使用 AsyncStorage 模块储存最近搜索的位置信息的功能。

例 6-10：SmarterWeather/weather_project.js

```
var React = require('react-native');
var {
  StyleSheet,
  Text,
  View,
  TextInput,
  AsyncStorage,
  Image
} = React;

var Forecast = require('./Forecast');
var LocationButton = require('./LocationButton');
var STORAGE_KEY = '@SmarterWeather:zip';
var WEATHER_API_KEY = 'bbeb34ebf60ad50f7893e7440a1e2b0b';
var API_STEM = 'http://api.openweathermap.org/data/2.5/weather?';

// 该版本使用flowers.png这个本地资源。
// var PhotoBackdrop = require('./PhotoBackdrop/local_image');

// 该版本允许你选择一张照片。
var PhotoBackdrop = require('./PhotoBackdrop');

// 该版本从相机选取一张特定的照片。
// var PhotoBackdrop = require('./PhotoBackdrop/camera_roll_example');

var WeatherProject = React.createClass({
  getInitialState () {
    return {
      forecast: null
    };
  },

  componentDidMount: function() {
    AsyncStorage.getItem(STORAGE_KEY)
      .then((value) => {
        if (value !== null) {
          this._getForecastForZip(value);
        }
      })
      .catch((error) => console.log('AsyncStorage error: ' + error.message))
      .done();
  },

  _getForecastForZip: function(zip) {
    // 储存邮编信息。
    AsyncStorage.setItem(STORAGE_KEY, zip)
      .then(() => console.log('Saved selection to disk: ' + zip))
```

```
      .catch((error) => console.log('AsyncStorage error: ' + error.message))
      .done();

    this._getForecast(
      `${API_STEM}q=${zip}&units=imperial&APPID=${WEATHER_API_KEY}`);
  },

  _getForecastForCoords: function(lat, lon) {
    this._getForecast(
      `${API_STEM}lat=${lat}&lon=${lon}&units=imperial&APPID=${WEATHER_API_KEY}`);
  },

  _getForecast: function(url, cb) {
    fetch(url)
      .then((response) => response.json())
      .then((responseJSON) => {
        console.log(responseJSON);
        this.setState({
          forecast: {
            main: responseJSON.weather[0].main,
            description: responseJSON.weather[0].description,
            temp: responseJSON.main.temp
          }
        });
      })
      .catch((error) => {
        console.warn(error);
      });
  },

  _handleTextChange: function(event) {
    var zip = event.nativeEvent.text;
    this._getForecastForZip(zip);
  },

  render: function() {
    var content = null;
    if (this.state.forecast !== null) {
      content = (
        <View style={styles.row}>
          <Forecast
            main={this.state.forecast.main}
            description={this.state.forecast.description}
            temp={this.state.forecast.temp}/>
        </View>);
    }

    return (
        <PhotoBackdrop>
            <View style={styles.overlay}>
              <View style={styles.row}>
                <Text style={textStyles.mainText}>
                  Current weather for
                </Text>
                <View style={styles.zipContainer}>
```

```
                    <TextInput
                      style={[textStyles.mainText, styles.zipCode]}
                      returnKeyType='go'
                      onSubmitEditing={this._handleTextChange}/>
                  </View>
                </View>
                <View style={styles.row}>
                  <LocationButton onGetCoords={this._getForecastForCoords}/>
                </View>
                {content}
              </View>
            </PhotoBackdrop>
        );
      }
    });

    var textStyles = require('./styles/typography.js');
    var styles = StyleSheet.create({
      overlay: {
        paddingTop: 5,
        backgroundColor: '#000000',
        opacity: 0.5,
      },
      row: {
        width: 400,
        flex: 1,
        flexDirection: 'row',
        flexWrap: 'nowrap',
        alignItems: 'center',
        justifyContent: 'center',
        padding: 30
      },
      zipContainer: {
        flex: 1,
        borderBottomColor: '#DDDDDD',
        borderBottomWidth: 1,
        marginLeft: 5,
        marginTop: 3,
        width: 10
      },
      zipCode: {
        width: 50,
        height: textStyles.baseFontSize,
      }
    });

  module.exports = WeatherProject;
```

它使用了 styles/typography.js 中的共享样式（例 6-11）。

例 6-11：共享字体样式在 SmarterWeather/styles/typography.js 中

```
var React = require('react-native');
var { StyleSheet } = React;
```

```
var baseFontSize = 18;

var styles = StyleSheet.create({
  bigText: {
    fontSize: baseFontSize + 8,
    color: '#FFFFFF'
  },
  mainText: {
    fontSize: baseFontSize,
    color: '#FFFFFF'
  }
});

// 允许在其他地方使用。
styles['baseFontSize'] = baseFontSize;

module.exports = styles;
```

6.4.2 Forecast组件

`<Forecast>` 组件展示了包括温度在内的预报信息，被其上的 `<WeatherProject>` 组件使用。该组件的代码见于例 6-12。

例 6-12：Forecast 组件渲染关于预报的信息

```
var React = require('react-native');
var { Text, View, StyleSheet } = React;
var styles = require('../styles/typography.js');

var Forecast = React.createClass({
  render: function() {
    return (
      <View style={forecastStyles.forecast}>
        <Text style={styles.bigText}>
          {this.props.main}
        </Text>
        <Text style={styles.mainText}>
          Current conditions:{this.props.description}
        </Text>
        <Text style={styles.bigText}>
          {this.props.temp}°F
        </Text>
      </View>
    );
  }
});

var forecastStyles = StyleSheet.create({
  forecast: {
    alignItems: 'center'
  }
});

module.exports = Forecast;
```

6.4.3 Button组件

<Button> 组件是一个可复用的容器样式组件。它提供了一个由 <TouchableHighlight> 包装的拥有合理样式的 <Text> 组件。组件的主要文件在例 6-13 中，关联的样式文件在例 6-14 中。

例 6-13：按钮组件提供一个拥有合理样式的由 <TouchableHighlight> 包装的 <Text> 组件

```
var React = require('react-native');
var {
  Text,
  View,
  TouchableHighlight
} = React;
var styles = require('./style.js');

var Button = React.createClass({
  propTypes: {
    onPress: React.PropTypes.func,
    label: React.PropTypes.string
  },

  render: function() {
    return (
      <TouchableHighlight onPress={this.props.onPress}>
        <View style={[styles.button, this.props.style]}>
          <Text>
            {this.props.label}
          </Text>
        </View>
      </TouchableHighlight>
      );
  }
});

module.exports = Button;
```

例 6-14：按钮组件的样式

```
var React = require('react-native');
var { StyleSheet } = React;

var baseFontSize = 16;

var styles = StyleSheet.create({
  button: {
    backgroundColor: '#FFDDFF',
    width: 200,
    padding: 25,
    borderRadius: 5
  },
});

module.exports = styles;
```

6.4.4 LocationButton组件

当被点击时，<LocationButton> 获取用户的位置信息并触发一个回调。组件的主 JavaScript 文件在例 6-15 中，样式文件在例 6-16 中。

例 6-15：<LocationButton> 组件

```
var React = require('react-native');
var styles = require('./style.js');
var Button = require('./../Button');

var LocationButton = React.createClass({
  propTypes: {
    onGetCoords: React.PropTypes.func.isRequired
  },

  _onPress: function() {
    navigator.geolocation.getCurrentPosition(
      (initialPosition) => {
        this.props.onGetCoords(initialPosition.coords.latitude,
          initialPosition.coords.longitude);
      },
      (error) => {alert(error.message)},
      {enableHighAccuracy: true, timeout: 20000, maximumAge: 1000}
    );
  },
  render: function() {
    return (
      <Button label="Use CurrentLocation"
        style={styles.locationButton}
        onPress={this._onPress}/>
      );
  }
});

module.exports = LocationButton;
```

例 6-16：<LocationButton> 的样式

```
var React = require('react-native');
var { StyleSheet } = React;

var baseFontSize = 16;

var styles = StyleSheet.create({
  locationButton: {
    backgroundColor: '#FFDDFF',
    width: 200,
    padding: 25,
    borderRadius: 5
   },
});

module.exports = styles;
```

6.4.5 PhotoBackdrop组件

为了说明选择背景图片的三种不同的方法，这里提供了三种版本的 `<PhotoBackdrop>` 组件。第一个版本见于例 6-17，在 GitHub 仓库里对应 local_image.js 文件，它使用 `require` 语句调用标准的图片资源。第二个版本见于例 6-18，对应 GitHub 仓库的 camera_roll_example.js 文件，它允许用户从相册里选择图片。最后，例 6-19 是第三个版本的实现，对应 GitHub 仓库的 index.js 文件，这个版本使用 `ImagePickerIOS` 来提示用户选择背景图片。

例 6-17：最初版本的 local_image.js，使用一个简单的 `require` 语句

```
var React = require('react-native');
var { Image } = React;
var styles = require('./style.js');

var PhotoBackdrop = React.createClass({
  render() {
    return (
        <Image
          style={styles.backdrop}
          source={require('image!flowers')}
          resizeMode='cover'>
          {this.props.children}
        </Image>
      );
  }
});

module.exports = PhotoBackdrop;
```

例 6-18：camera_roll_example.js 使用代码从相册中选择一张图片

```
var React = require('react-native');
var { Image, CameraRoll } = React;
var styles = require('./style.js');

var PhotoBackdrop = React.createClass({
  getInitialState() {
    return {
      photoSource: null
    }
  },
  componentDidMount() {
    CameraRoll.getPhotos(
      {first: 5},
      (data) => {
        this.setState({
          photoSource: {uri: data.edges[3].node.image.uri}
        })},
      (error) => {
        console.warn(error);
      });
  },
  render() {
```

```
    return (
      <Image
        style={styles.backdrop}
        source={ this.state.photoSource }
        resizeMode='cover'>
        {this.props.children}
      </Image>
      );
  }
});

module.exports = PhotoBackdrop;
```

例 6-19：最后一个版本的 index.js，采用 `ImagePickerIOS` 并提示用户选择一张图片

```
var React = require('react-native');
var {
  Image,
  ImagePickerIOS
} = React;
var styles = require('./style.js');

var Button = require('./../Button');

var PhotoBackdrop = React.createClass({
  getInitialState() {
    return {
      photoSource: require('image!flowers')
    }
  },
  _pickImage() {
    ImagePickerIOS.openCameraDialog(
      {},
      (data) => {
        this.setState({
          photoSource: {uri: data}
        });
      },
      () => {
        console.log('User canceled the action');
      });
  },
  render() {
    return (
      <Image
        style={styles.backdrop}
        source={ this.state.photoSource }
        resizeMode='cover'>
        {this.props.children}
        <Button
          style={styles.button}
          label="Load Image"
          onPress={this._pickImage}/>
      </Image>
      );
```

```
  }
});

module.exports = PhotoBackdrop;
```

这三个版本都共享了相同的样式，如例 6-20 所示。

例 6-20：三个版本的 `<PhotoBackdrop>` 都使用这个样式表

```
var React = require('react-native');
var { StyleSheet } = React;

var styles = StyleSheet.create({
  backdrop: {
    flex: 1,
    flexDirection: 'column'
  },
  button: {
    flex: 1,
    margin: 100,
    alignItems: 'center'
  }
});

module.exports = styles;
```

6.5　小结

在本章中，我们对天气应用进行了一些修改，然后接触了地理定位（Geolocation）、相机（Camera Roll）和存储（`AsyncStorage`）接口，最后学习了如何将这些接口整合到应用中去。由于这些接口的支持程度因平台而不同，所以我们应该隔离将要使用的组件，由此我们就可以包装平台无关的组件，正如第 4 章所展示的。

撇开兼容性问题，如果 React Native 提供了宿主平台接口的支持，那么使用起来就会得心应手。但是假如 React Native 不支持某个接口，比如视频回放的功能，这时你需要使用一个非 JavaScript 实现的类库或模块，那应该怎么做呢？在下一章中，我们将详细介绍这种情况的解决方案。

第 7 章

模块

第 6 章介绍了一些如何与 React Native 封装的宿主平台接口交互的知识。例如，相机和地理定位这样的接口是平台特定的，但 React Native 为了便利，已经帮我们暴露了接口。由于这些接口已经集成到 React Native，因此使用起来非常方便。

如果要使用一个 React Native 不支持的接口，应该怎么做呢？本章将介绍如何通过 npm 安装由 React Native 社区编写的模块。同时，我们也要深入学习一个 iOS 模块 `react-native-video` 的安装过程，以及 `RCTBridgeModule` 是如何允许用户为现有的 Objective-C 接口添加 JavaScript 接口的。最后，我们再来看如何导入纯 JavaScripto 类库到你的工程中，以及如何管理依赖。

本章会涉及一些 Objective-C 和 Java 代码，不过别担心，我们会放慢脚步。完整的 iOS 和 Android 移动应用开发的介绍超出了本书的范围，但是我们也会一起学习一些例子。

7.1　使用npm安装JavaScript类库

在讨论原生模块工作原理之前，先来看外部依赖通常是怎样安装的。React Native 使用 npm 进行依赖管理。npm 虽然是一个 Node.js 的包管理器，但 npm 仓库囊括了所有 JavaScript 工程，而不仅仅是 Node 平台。npm 使用一个名为 package.json 的文件来储存包括一系列依赖在内的项目元数据。

我们从一个全新的项目开始：

```
react-native init Depends
```

创建一个新工程之后，你的 package.json 文件看起来如下所示：

```
{
  "name": "Depends",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node_modules/react-native/packager/packager.sh"
  },
  "dependencies": {
    "react-native": "^0.12.0"
  }
}
```

目前为止，项目最顶层的依赖是 `react-native`。让我们添加其他的依赖！

lodash 库与 Underscore.js 非常类似，它提供了一系列实用的工具函数，例如针对数组的 `shuffle` 函数。我们在安装时附带 `--save` 参数，它将会被添加到依赖列表中：

```
npm install --save lodash
```

现在，你的 package.json 文件被更新如下：

```
"dependencies": {
  "lodash": "^3.10.1",
  "react-native": "^0.12.0"
}
```

如果想在 React Native 应用中使用 lodash，可以这样引入它：

```
var _ = require('lodash');
```

现在，我们使用 lodash 来打印一个随机数：

```
var _ = require('lodash');
console.log('Random number: ' + _.random(0, 5));
```

成功输出了！但是其他模块是怎样的呢？我们可以通过 `npm install` 引入任何第三方包吗？

答案当然是"可以"，但有一些额外的说明。例如，任何涉及 DOM 操作的方法都会运行失败。和现有的第三方包集成可能需要一些技巧，因为大多数类库都假定了它们运行的环境。但一般而言你可以使用任何 JavaScript 包，也可以和其他 JavaScript 工程一样使用 npm 来管理项目依赖。

7.2 iOS原生模块

既然已经学会了如何通过 npm 添加外部 JavaScript 类库，那么现在让我们通过 npm 安装一个 React Native 组件吧。在这一节中，我们尝试使用 `react-native-video`，一个由

Brent Vatne 实现的 React Native 组件，把它作为主要的例子。这个模块为我们提供了一个 `<Video>` 组件，可以用来播放视频（太棒了！）。最后，我们将会学习底层原生模块与 Objective-C 及 iOS 协同工作的原理。

7.2.1　导入第三方组件

`react-native-video` 组件储存在 npm 仓库（https://www.npmjs.com/package/react-native-video）中。我们可以通过 `npm install` 将其添加到项目里：

```
npm install react-native-video --save
```

如果是在做传统的 Web 开发，那么一切都已经完成了！ `react-native-video` 已经可以在项目中使用了。不幸的是，在这里并不是这样的；在 iOS 开发中，我们需要告知 Xcode 存在这个类库。

在 Xcode 中打开项目，在 Libraries 上单击右键，然后选择 Add Files to“Depends”...（图 7-1）。

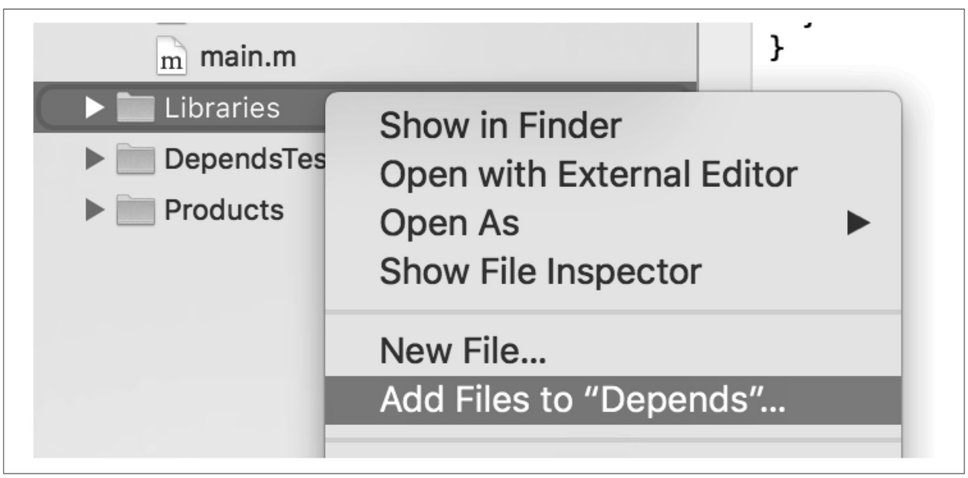


图 7-1：右键单击 Libraries，选择 Add Files to “Depends” ...

接着，添加 RCTVideo.xcodeproj 文件到工程中（图 7-2）。

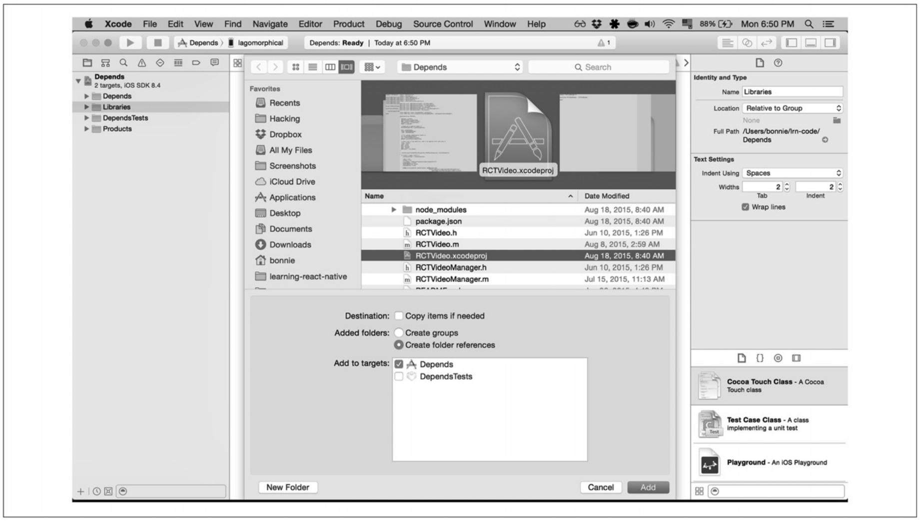


图 7-2：从列表中选择 RCTVideo.xcodeproj；它应该在 node_modules/react-native-video 目录下

同时，你也需要添加视频框架到工程的 build process（构建过程）中。打开工程的 build process，然后展开 Link Binary With Libraries 子菜单，点击“+”号，添加 libRCTVideo.a 到工程中（图 7-3）。

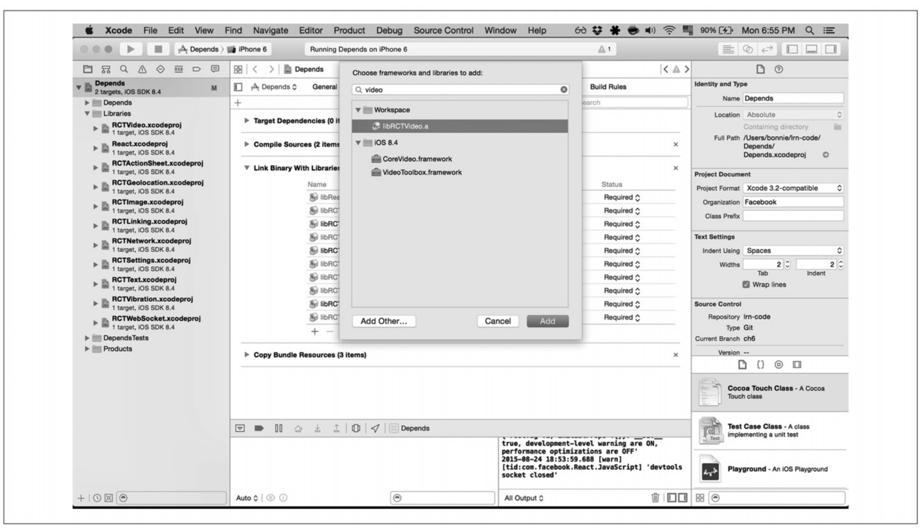


图 7-3：添加 libRCTVideo.a 文件（可以使用搜索框帮助你定位文件）

完成上面的工作之后，RCTVideo 模块就成功导入到工程中了。我们还需要导入一个 mp4 视频文件到 Xcode 工程，这样才可以把它当做资源使用。在工程文件上单击右键，再次选择 Add Files to Depends，如图 7-4 所示。

任何 mp4 视频文件应该都可以使用。我使用的是手头已有的一个项目中的视频，你可以从 GitHub 网站下载（https://github.com/bonniee/learning-react-native/blob/master/Depends/iOS/PianoStairs.mp4?raw=true）。

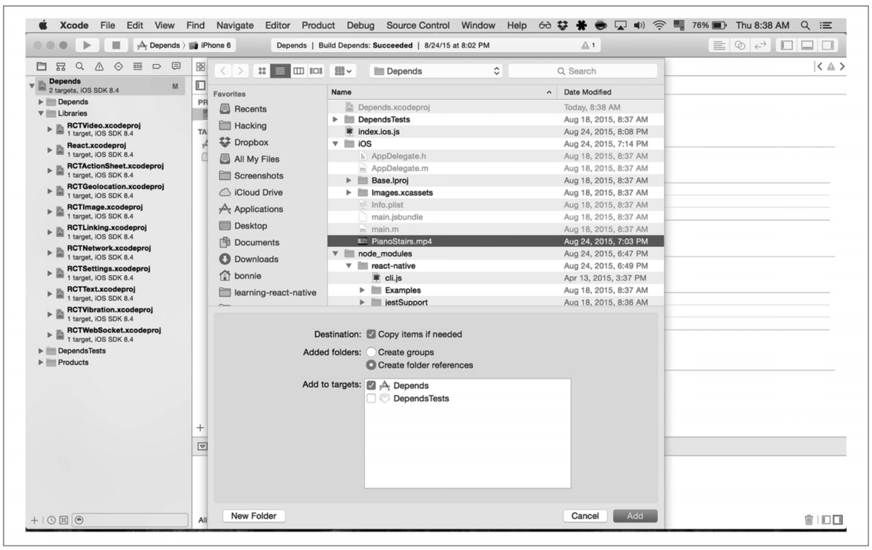


图 7-4：选择想使用的视频文件；这里，我们选择 PianoStairs.mp4 文件

然后，你应该可以在工程列表里看到一个视频文件（图 7-5）。

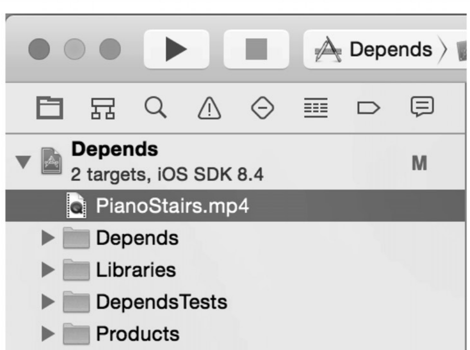


图 7-5：视频被成功添加到工程之后，可以在 Xcode 中看到它

7.2.2　使用视频组件

好了，既然我们已经成功导入到 Xcode，现在可以在 JavaScript 代码中引入 Video 组件了。

```
var Video = require('react-native-video');
```

然后你就可以像使用普通组件一样使用它了。这里我设置了一些可选的属性：

```
<Video source={{uri: "PianoStairs"}} // 可以是URL或本地文件。
       rate={1.0}                    // 0 表示暂停,1 表示正常。
       volume={1.0}                  // 0 表示静音,1 表示正常。
       muted={false}                 // 是否完全静音。
       paused={false}                // 是否完全暂停播放。
       resizeMode="cover"            // 是否按高宽比覆盖整个屏幕。
       repeat={true}                 // 是否自动循环播放。
       style={styles.backgroundVideo} />
```

哇！我们添加了一个视频组件！

虽然相比于简单的 `npm install` 命令，使用 React Native 第三方模块的过程可能有些复杂，但还可以接受。比较让人困惑的应该就是引入第三方库到 Xcode 工程以及怎样使用 Xcode 了。像 `react-native-video` 这样的模块通常都是专为 React Native 开发的，在它项目的 README 文件里一般都有详细的安装和使用说明，所以这应该不成问题。不要让 Xcode 的这部分交互阻碍你在代码里使用第三方模块！

还有大量这样的组件都在 npm 仓库里，它们通常都使用了 `react-native-` 前缀。你可以随便逛逛，看看社区都开放了哪些资源。

7.2.3　剖析Objective-C原生模块

既然我们使用过了 `react-native-video` 组件，现在我们一起看看模块在底层究竟是怎样工作的吧。

`react-native-video` 是一个 React 引用原生模块的组件（https://facebook.github.io/react-native/docs/native-modules-ios.html）。React Native 文档是这样定义原生模块的："一个实现了 `RCTBridgeModule` 协议的 Objective-C 类。"（RCT 是 ReaCT 的缩写。）

编写 Objective-C 代码不是 React Native 标准开发流程的一部分，所以不用担心，这不是必要的知识！当然，如果有基础的代码阅读能力，可以看懂代码的话，会对你有很大的帮助，即便你还没有打算实现自己的原生模块。

如果你之前从未使用过 Objective-C，可能大多数的语法会让你感到困惑。没关系，慢慢来。首先我们开发一个基础的"Hello, World"模块。

Objective-C 类通常有一个以 .h 结尾的头文件，它包含了类的接口。但实际上我们在 .m 里

实现功能。我们从编写 HelloWorld.h 文件开始，如例 7-1 所示。

例 7-1：Depends/iOS/HelloWorld.h

```
#import "RCTBridgeModule.h"

@interface HelloWorld : NSObject <RCTBridgeModule>
@end
```

这个文件有什么用处呢？在第一行，我们导入了 `RCTBridgeModule` 头文件（注意：# 符号不表示注释，而是导入语句的一部分）。接着下一行，我们定义了一个 `HelloWorld` 类，它继承自 `NSObject` 并实现了 `RCTBridgeModule` 接口，最后通过 `@end` 完成接口的定义。

由于历史原因，许多 Objective-C 的类型都有 `NS` 前缀（`NSString`、`NSObject` 等）。

现在我们转移到具体的实现中去（例 7-2）。

例 7-2：Depends/iOS/HelloWorld.m

```
#import "HelloWorld.h"
#import "RCTLog.h"

@implementation HelloWorld

RCT_EXPORT_MODULE();

RCT_EXPORT_METHOD(greeting:(NSString *)name)
{
  RCTLogInfo(@"Saluton, %@", name);
}

@end
```

在一个 .m 文件里，你需要导入相应的 .h 文件，像第一行这样。我还导入了 RCTLog.h，这样我们就可以使用 `RCTLogInfo` 输出日志到控制台。在导入其他 Objective-C 类时，我们总是导入头文件，而不是 .m 文件。

`@implementation` 和 `@end` 这两行表明在它们之间的内容是 `HelloWorld` 类的具体实现。

剩下的几行是 React Native 模块主要功能的实现代码。通过 `RCT_EXPORT_MODULE()`，我们调用了一个特殊的 React Native 宏，由此可以访问 React Native 的“桥接”。同样，我们的 `greeting:name` 的方法定义也以一个宏开头，它导出了 `RCT_EXPORT_METHOD` 这一方法，因此我们可以在 JavaScript 代码中使用它。

需要注意的是，这里 Objective-C 方法的命名有一点奇怪。每一个参数名称都是被包含在方法名称里的。这是 React Native 的约定，JavaScript 函数名称是 Objective-C 名称从开始到第一个冒号为止的部分，因此 `greeting:name` 就成了 JavaScript 里的 `greeting`。如果你愿意，可以使用 `RCT_REMAP_METHOD` 宏重新制定命名规则。

然后就可以在 JavaScript 文件里调用这个方法了（例 7-3）。

例 7-3：在 JavaScript 代码里使用 `HelloWorld` 模块

```
var HelloWorld = require('react-native').NativeModules.HelloWorld;
HelloWorld.greeting('Bonnie');
```

输出的内容应该会出现在控制台里（图 7-6），你可以同时在 Xcode 和 Chrome 开发者工具里看到输出内容，如果同时启用了它们的话。

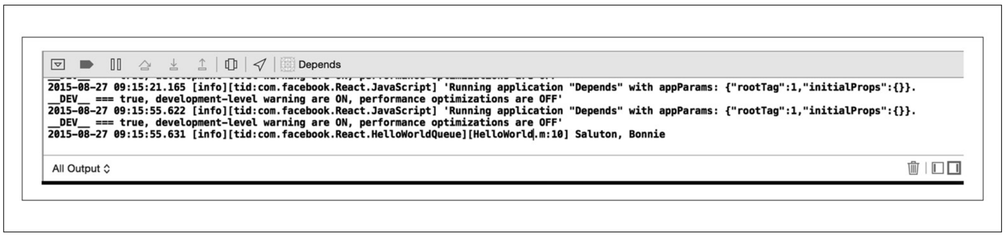


图 7-6：控制台输出，通过 Xcode 界面查看

可以看到导入原生模块的语法有点冗长。一个常用的办法是用 JavaScript 封装原生模块（例 7-4）。

例 7-4：Depends/HelloWorld.js：用 JavaScript 封装 HelloWorld 原生模块

```
var HelloWorld = require('react-native').NativeModules.HelloWorld;
module.exports = HelloWorld;
```

于是，导入过程变得更加直观了：

```
var HelloWorld = require('./HelloWorld');
```

HelloWorld.js 这个 JavaScript 文件也是一个添加 JavaScript 代码到你的模块中的一个好方法。

哇，Objective-C 让人感觉有些冗长，并且我们需要操作几个不同的文件。不过要祝贺你，你已经用 Objective-C 编写了一个“Hello, World”模块啦！

回顾一下，使一个 Objective-C 模块生效，需要遵守以下几条规则：

- 导入 `RCTBridgeModule` 头文件；
- 声明模块并实现 RCTBridgeModule 协议；
- 调用 `RCT_EXPORT_MODULE()` 宏；
- 使用 `RCT_EXPORT_METHOD` 宏导出至少一个方法。

然后原生模块就可以使用 iOS SDK 提供的任何接口了（注意：你为 React Native 提供的接口必须是异步的）。Apple 公司提供了 iOS SDK 的扩展文档，同时还有大量的第三方资源

可以使用。值得一提的是，你的开发者账号现在就会派上用场，通常没有开发者账号就很难访问 SDK 文档。

既然我们已经编写了自己的基础“Hello, World”模块，那么下一步将深入介绍 react-native-video 是如何工作的。

7.2.4 RCTVideo的实现

正如 HelloWorld 模块一样，RCTVideo 也是一个原生的模块，并且它实现了 RCTBridgeModule 协议。你可以在 GitHub 仓库看到 RCTVideo 完整的代码（https://github.com/brentvatne/react-native-video），我们使用的是 0.6.0 版本。

react-native-video 是一个由 iOS SDK 提供的 AVPlayer 接口的一个基础封装。我们再具体看看它是如何工作的，从 Video.ios.js 和 Video.android.js 这两个 JavaScript 入口文件开始。在这个版本里，Video.android.js 还没有被实现，所以我们看看 Video.ios.js（https://github.com/brentvatne/react-native-video/blob/1f0ba1347b783c2f1e0fd36a2a757560f296ace5/Video.ios.js）。

我们会发现它为原生组件提供了一个很薄的封装层，RCTVideo 执行一些属性的规范化检查，还有一些额外的渲染逻辑。原生组件在末尾被导入：

```
var RCTVideo = requireNativeComponent('RCTVideo', Video);
```

正如在 HelloWorld 例子中看到的一样，这里意味着 RCTVideo 组件一定在 Objective-C 的某处被导出了。我们看看 RCTVideo.h（https://github.com/brentvatne/react-native-video/blob/ 1f0ba1347b783c2f1e0fd36a2a757560f296ace5/RCTVideo.h）：

```
// RCTVideo.h
#import "RCTView.h"

@class RCTEventDispatcher;

@interface RCTVideo : UIView

- (instancetype)initWithEventDispatcher:
(RCTEventDispatcher *)eventDispatcher NS_DESIGNATED_INITIALIZER;

@end
```

这一次，它没有继承自 NSObject，而是继承了 UIView。这很容易理解，因为它渲染了一个视图组件。

如果再来看实现的文件 RCTVideo.m（https://github.com/brentvatne/react-native-video/blob/1f0ba1347b783c2f1e0fd36a2a757560f296ace5/RCTVideo.m），会发现这里有很多代码。在实例变量的顶部，记录了音量、播放速率和 AVPlayer 自身等信息。

这里应该看一下 methodQueue 这个有趣的方法：

```
- (dispatch_queue_t)methodQueue
{
  return dispatch_get_main_queue();
}
```

这里告知系统必须使用 iOS 的主线程执行代码，因为该模块使用了仅适用于主线程的 iOS 接口。

代码里还有一些方法，比如计算视频的长度、加载视频和设置源及更多的功能。不妨看一下这些方法，了解它们起了什么作用。

另一个让人困惑的是 RCTVideoManager。为了创建一个原生 UI 组件而不仅仅是一个模块，我们需要一个视图管理器。正如它的名称所表示的，当视图在处理一些渲染逻辑和类似的任务时，视图管理器就会处理其他的逻辑（事件处理、属性导出等）。视图管理器类至少需要：

- 继承 RCTViewManager 类；
- 使用 RCT_EXPORT_MODULE() 宏；
- 实现 -(UIView *)view 方法。

view 方法需要返回一个 UIView 实例。这里我们发现它被实例化并返回了一个 RCTVideo：

```
- (UIView *)view
{
  return [[RCTVideo alloc] initWithEventDispatcher:self.bridge.eventDispatcher];
}
```

RCTVideoManager 也导出了一些属性和常量：

```
RCT_EXPORT_VIEW_PROPERTY(src, NSDictionary);
RCT_EXPORT_VIEW_PROPERTY(resizeMode, NSString);
RCT_EXPORT_VIEW_PROPERTY(repeat, BOOL);
RCT_EXPORT_VIEW_PROPERTY(paused, BOOL);
RCT_EXPORT_VIEW_PROPERTY(muted, BOOL);
RCT_EXPORT_VIEW_PROPERTY(volume, float);
RCT_EXPORT_VIEW_PROPERTY(rate, float);
RCT_EXPORT_VIEW_PROPERTY(seek, float);

- (NSDictionary *)constantsToExport
{
  return @{
    @"ScaleNone": AVLayerVideoGravityResizeAspect,
    @"ScaleToFill": AVLayerVideoGravityResize,
    @"ScaleAspectFit": AVLayerVideoGravityResizeAspect,
    @"ScaleAspectFill": AVLayerVideoGravityResizeAspectFill
  };
```

`RCTVideo` 和 `RCTVideoManager` 共同由 `RCTVideo` 原生 UI 组件组成，我们可以在应用内随意使用它。正如你所看到的，利用 iOS SDK 编写原生模块需要花费一些工夫，但也不是不可驾驭的。你之前的 iOS 开发经验会对你有很大的帮助。iOS 开发完整的介绍目前超出了本书的范围，但即使你没有 Objective-C 的经验，通过查看别人的原生模块，你也应该能开始尝试编写自己的原生模块了。

7.3 Android原生模块

Android 原生模块与 iOS 原生模块非常类似。你可以在官方文档中找到更多关于 Android 原生模块的相关信息（https://facebook.github.io/react-native/docs/native-modules-android.html）。

在这一节中，我们将在 AndroidDepends/ 项目目录下进行开发。首先进入目录，然后创建一个新的项目：

```
react-native init AndroidDepends
```

7.3.1 安装第三方组件

此处将安装 `react-native-linear-gradient` 包，它为我们提供了一个 `<LinearGradient>` 组件。由于创建渐变是一个相对依赖显卡的任务，所以这个组件采用原生平台接口是非常有意义的。为此，`<LinearGradient>` 提供了统一的 React Native 组件，它分别在底层各自调用了 Android 的 `android.graphics` 包和 iOS 的 `CAGradientLayer` 接口。你可以在 GitHub 上找到这个项目（https://github.com/brentvatne/react-native-linear-gradient）。

正如 iOS 平台一样，安装 Android 平台的原生模块也需要接触 Android 项目的代码。总体来说，为了导入一个第三方 Android 原生模块，需要以下三个步骤：

(1) 更新 android/settings.gradle 文件，将模块加入到 Android 构建过程中；
(2) 把模块作为依赖列在 android/app/build.gradle 文件内；
(3) 在 MainActivity.java 里导入包，并将它引入成为一个 React Native 可用的包。

让我们来逐一查看这些步骤。首先更新 settings.gradle 文件，导入 react-native-linear-gradient 目录。settings.gradle 文件如例 7-5 所示。

例 7-5：AndroidDepends/android/settings.gradle

```
rootProject.name = 'AndroidDepends'

include ':app', ':react-native-linear-gradient'
project(':react-native-linear-gradient').projectDir =
  new File(rootProject.projectDir,
    '../node_modules/react-native-linear-gradient/android')
```

“gradle”是一个 Android 的构建系统。当我们使用 `npm install` 安装 `react-native-linear-gradient` 包时，相关的 Android 文件也会被下载到 node_modules/ 目录下。我们通过更新 settings.gradle 文件来引入这个目录。

接下来，我们要把 `react-native-linear-gradient` 模块作为依赖添加到 build.gradle 文件中（例 7-6）。你会发现这个文件已经包含了一些构建设置，例如目标 Android SDK 的版本以及 React Native 这样的应用依赖。我们要把 `react-native-linear-gradient` 加入到文件末尾的依赖列表中。

例 7-6：AndroidDepends/android/app/build.gradle

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.1"

    defaultConfig {
        applicationId "com.androiddepends"
        minSdkVersion 16
        targetSdkVersion 22
        versionCode 1
        versionName "1.0"
        ndk {
            abiFilters "armeabi-v7a", "x86"
        }
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
            'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:23.0.1'
    compile 'com.facebook.react:react-native:0.12.+'
    compile project(':react-native-linear-gradient')
}
```

最后，我们要更新 MainActivity.java 文件，有两个步骤：导入 `LinearGradientPackage`，然后添加到 `ReactInstanceManager` 中。

你可以添加以下语句到文件头部的任意位置，只要在类定义之前即可。

```
import com.BV.LinearGradient.LinearGradientPackage;
```

接着，通过在现有的语句之后添加一个 addPackage() 语句，我们将包添加到 ReactInstanceManager 里。

```
mReactInstanceManager = ReactInstanceManager.builder()
        .setApplication(getApplication())
        .setBundleAssetName("index.android.bundle")
        .setJSMainModuleName("index.android")
        .addPackage(new MainReactPackage())
        .addPackage(new LinearGradientPackage()) // 添加这行!
        .setUseDeveloperSupport(BuildConfig.DEBUG)
        .setInitialLifecycleState(LifecycleState.RESUMED)
        .build();
```

好了！这个步骤完成之后，我们可以在 JavaScript 中导入这个包了，代码如下：

```
var LinearGradient = require('react-native-linear-gradient');
```

然后可以在 React Native 中使用这个组件：

```
 <LinearGradient colors={['#FFFFFF', '#00A8A8']} style={styles.container}>
   <Text style={styles.welcome}>
     A Lovely Gradient
   </Text>
</LinearGradient>
```

我们使用它创建一个 <Gradient> 组件，来代替默认的应用界面（例 7-7）。

例 7-7：AndroidDepends/gradient.js

```
var React = require('react-native');
var {
  StyleSheet,
  Text
} = React;
var LinearGradient = require('react-native-linear-gradient');

var Gradient = React.createClass({
  render: function() {
    return (
        <LinearGradient colors={['#FFFFFF', '#00A8A8']} style={styles.container}>
          <Text style={styles.welcome}>
            A Lovely Gradient
          </Text>
        </LinearGradient>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
```

```
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
    height: 50,
    padding: 20
  }
});

module.exports = Gradient;
```

这就是使用 `<LinearGradient>` 组件需要做的所有工作。修改 index.android.js 文件来渲染一个 `<Gradient>` 组件，最终它会渲染一个带文本的渐变，如图 7-7 所示。

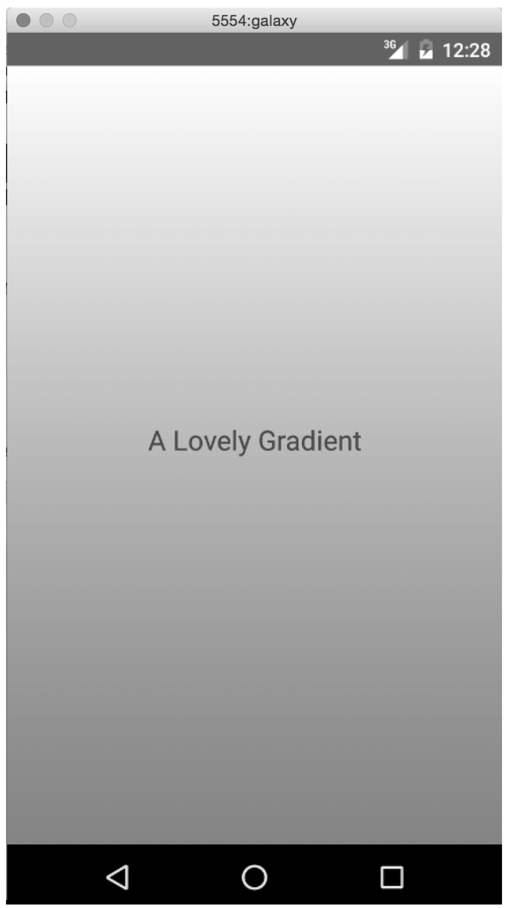


图 7-7：`<Gradient>` 组件

太酷了！既然我们已经学习了如何在 Android 中引入一个第三方模块，那我们再创建一个“Hello, World”模块来看看原生模块的大致工作原理，但这次我们使用 Java 代替 Objective-C。最后，我们再深入剖析一下 `react-native-linear-gradient` 的内部原理。

7.3.2 剖析Java原生模块

为了更好地了解 Java 原生模块的工作原理，我们将自己动手编写一个。就像 Objective-C 中那样，我们从简单的“Hello, World”模块开始。

首先，创建一个 HelloWorld.java 文件（例 7-8）。记住，Android 项目有相当深的嵌套结构。我们把 HelloWorld.java 和 MainActivity.java 放在同级目录下。

例 7-8：AndroidDepends/android/app/src/main/java/com/androiddepends/ HelloWorld.java

```
package com.androiddepends;

import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;

import android.util.Log;

public class HelloWorld extends ReactContextBaseJavaModule {

  public HelloWorld(ReactApplicationContext reactContext) {
    super(reactContext);
  }

  @Override
  public String getName() {
    return "HelloWorld";
  }

  @ReactMethod
  public void greeting(String name) {
    Log.i("HelloWorld", "Hello, " + name);
  }
}
```

这是一个相当不错的模板！让我们一步步来学习。

首先，从 `package` 语句开始：

```
package com.androiddepends;
```

所有在 `com.androiddepends` 包下的文件都必须以这一行开头。出于方便考虑，我们使用与 MainActivity.java 文件相同的包。

接着导入 React Native 特定的文件以及 android.util.Log。任何编写的模块都需要导入相同的 React Native 文件。

然后定义一个 HelloWorld 类。它是公开的，这意味着外部文件可以使用它。并且它继承了 ReactContextBaseJavaModule，从而也继承了 ReactContextBaseJavaModule 中的方法：

```
public class HelloWorld extends ReactContextBaseJavaModule { ... }
```

这里实现了三个方法：HelloWorld、getName 和 greeting。

在 Java 中，与类名名称相同的方法称作*构造方法*。因此，HelloWorld 方法有点像模板；我们只需调用 super(reactContext) 方法就可以调用 ReactContextBaseJavaModule 的构造方法，不需要额外的工作。

getName 决定了今后我们将在 JavaScript 里使用什么名称来调用这个模块，因此需要确保它是正确的！在我们的例子中，模块命名为“HelloWorld”。注意，这里我们添加了一个 @Override 注解。无论编写什么模块，你都需要实现 getName 方法。

最后，greeting 是我们自己的方法，我们希望能在 JavaScript 代码中调用它。我们添加了一个 @ReactMethod 注解，为了让 React Native 知道这个方法应该被导出。为了在调用 greeting 方法时添加日志，我们调用了 Log.i，操作如下：

```
Log.i("HelloWorld", "Hello, " + name);
```

Android 中的 Log 对象提供了不同层级的日志功能，其中三个最常用的功能是：消息（INFO）、警告（WARN）和错误（ERROR），它们各自通过 Log.i、Log.w 和 Log.e 方法进行调用。每个方法都接收两个参数，一个是日志的标签名，另一个是日志信息。标准的做法是使用类名作为标签名。你可以通过 Android 文档查看更多细节（http://developer.android.com/reference/android/util/Log.html）。

我们还需要创建一个包文件来包装这个模块（例 7-9），这样我们可以将它加入构建过程中。该文件也应该与 HelloWorld.java 文件处于同级目录下。

例 7-9：AndroidDepends/android/app/src/main/java/com/androiddepends/ HelloWorldPackage.java

```
package com.androiddepends;

import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.JavaScriptModule;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
```

```
public class HelloWorldPackage implements ReactPackage {
  @Override
  public List<NativeModule>
  createNativeModules(ReactApplicationContext reactContext) {
    List<NativeModule> modules = new ArrayList<>();
    modules.add(new HelloWorld(reactContext));
    return modules;
  }

  public List<Class<? extends JavaScriptModule>> createJSModules() {
    return Collections.emptyList();
  }

  public List<ViewManager> createViewManagers(ReactApplicationContext reactContext) {
    return Collections.emptyList();
  }
}
```

这个文件基本上就是一个模板。我们不需要导入 `HelloWorld`，因为它们在同一个包下（`com.androiddepends`）。这里有三个方法需要注意：`createNativeModules`、`createJSModules` 和 `createViewManagers`。React Native 使用这些方法来决定需要导出哪些模块。

在这里我们只是编写一个简单的所谓的原生模块，因此后两个方法返回了一个空的列表，而 `createNativeModules` 方法返回了一个包含 `HelloWorld` 实例的列表。相反，如果查看 LinearGradientPackage.java 文件（源码：https://github.com/brentvatne/react-native-linear-gradient/blob/master/android/src/main/java/com/BV/LinearGradient/LinearGradientPackage.java），你会发现它在 `createViewManagers` 方法中返回了一个 `LinearGradientManager` 的实例，而其他两个方法返回了空列表。

最后，我们需要在 MainActivity.java 里添加包，就像之前导入 `LinearGradient` 的做法一样。导入包文件：

```
import com.androiddepends.HelloWorldPackage;
```

接着添加 `HelloWorldPackage` 到 `ReactInstanceManager` 中：

```
mReactInstanceManager = ReactInstanceManager.builder()
        .setApplication(getApplication())
        .setBundleAssetName("index.android.bundle")
        .setJSMainModuleName("index.android")
        .addPackage(new MainReactPackage())
        .addPackage(new LinearGradientPackage())
        .addPackage(new HelloWorldPackage()) // <-- Add this line
        .setUseDeveloperSupport(BuildConfig.DEBUG)
        .setInitialLifecycleState(LifecycleState.RESUMED)
        .build();
```

就像 Objective-C 模块一样，我们也是通过 `React.NativeModules` 对象引入 Java 模块。现在，我们可以在应用的任何地方调用 `greeting()` 方法，就像这样：

```
React.NativeModules.HelloWorld.greeting("Bonnie");
```

让我们过滤日志来查看信息。在项目的根目录下运行命令：

```
adb logcat | grep HelloWorld
```

我们查找“HelloWorld”实例，因为它是我们在 `Log.i` 里使用的标签名。图 7-8 就是应该在终端中看到的输出内容。

```
10-11 14:01:45.081  2335  2369 I HelloWorld: Hello, Bonnie
10-11 14:01:45.081  2335  2369 I HelloWorld: Hello, Bonnie
```

图 7-8：logcat 的输出内容

既然已经使用 Java 编写了一个“Hello, World”模块，那我们再看一个更复杂的例子：`react-native-linear-gradient`。

7.3.3 LinearGradient的Android实现

`<LinearGradient>` 的 Android 实现的代码在 android/ 目录下（https://github.com/brentvatne/react-native-linear-gradient/tree/master/android）。它主要包含了以下三个文件：

- LinearGradientPackage.java
- LinearGradientView.java
- LinearGradientManager.java

LinearGradientPackage.java，如例 7-10 所示，看起来跟 HelloWorldPackage.java 文件非常相似。

例 7-10：LinearGradientPackage.java

```
package com.BV.LinearGradient;

import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.JavaScriptModule;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class LinearGradientPackage implements ReactPackage {
```

```
    @Override
    public List<NativeModule>
    createNativeModules(ReactApplicationContext reactContext) {
      return Collections.emptyList();
    }

    public List<Class<? extends JavaScriptModule>> createJSModules() {
      return Collections.emptyList();
    }

    public List<ViewManager> createViewManagers(ReactApplicationContext reactContext) {
      List<ViewManager> modules = new ArrayList<>();
      modules.add(new LinearGradientManager());
      return modules;
    }
  }
```

主要的不同是 LinearGradientPackage 从 createViewManagers 方法返回了 LinearGradient-Manager 实例，而我们的 HelloWorldPackage 通过 createNativeModules 方法返回 Hello-World 实例。

对于 Android 而言，任何原生渲染的视图都是由 ViewManager 创建和控制的（或者更具体来说，是继承了 ViewManager 的类）。由于 LinearGradient 是一个 UI 组件，我们要返回一个 ViewManager 实例。React Native 文档中的 Android UI 组件部分（https://facebook.github.io/react-native/docs/native-components-android.html）有更多关于暴露原生接口（即非渲染的 Java 代码）和 UI 组件的区别的内容。

让我们再看看 LinearGradientManager。这是一个相当长的文件，你可以在 react-native-linear-gradient 项目的 GitHub 仓库查看完整的源码（https://github.com/brentvatne/react-native-linear-gradient）。我们在此查看一个缩略的版本：

```
public class LinearGradientManager extends SimpleViewManager<FrameLayout> {

  ...

  public static final String REACT_CLASS = "BVLinearGradient";
  public static final String PROP_COLORS = "colors";
  public LinearGradientView mGradientView;

  ...

  @Override
  public String getName() {
      return REACT_CLASS;
  }

  ...

  @ReactProp(name=PROP_COLORS)
```

```
    public void updateColors(FrameLayout frame, ReadableArray colors){
        if(mGradientView != null) {
            mGradientView.updateColors(colors);
        }
    }

    ...

    @Override
    public void updateView(FrameLayout frame, CatalystStylesDiffMap props) {
        BaseViewPropertyApplicator.applyCommonViewProperties(frame, props);

        frame.removeAllViews();

        mGradientView = new LinearGradientView(frame.getContext(), props);
        mGradientView.setId(View.generateViewId());

        frame.addView(mGradientView);
    }
}
```

这里有一些我们需要注意的地方。

首先是 getName。应该引起注意的是，就像我们的 HelloWorld 例子一样，实现 getName 方法是为了让 JavaScript 代码可以引用这个组件。

下一个需要注意的是 updateColors 方法，以及 @ReactProp 注解的使用。这里，我们定义了 <LinearGradient> 组件接收一个 colors 属性名（因为这是 PROP_COLORS 的值），并且当属性改变时，updateColors 方法会被调用。在 updateColors 方法中，我们检查基础视图是否存在；如果存在，那么我们传入颜色，这样它就可以被更新了。

最后，在 updateView 中，LinearGradientManager 实际上处理视图的更新逻辑，从帧布局中移除已存在的视图，然后添加一个新的 LinearGradientView 实例到帧布局中。

为了高效地编写 Android 组件，你需要了解 Android 大体上是如何处理视图的，但查看其他 React Native 组件源码也是一个很好的做法。

7.4 跨平台原生模块

编写一个跨平台原生模块是可能的吗？

答案当然是“可能”。你只需要分别为每一个平台都实现一个模块，然后对外提供统一的 JavaScript 接口即可。这个方法可以在代码复用最大化的前提下很好地解决特定平台的优化问题。

<LinearGradient> 组件就是一个很好的例子。我们的 AndroidDepends 项目是真正跨平台的，因为 <LinearGradient> 组件是跨平台渲染的（即便有一些样式不相同），如图 7-9 所示。

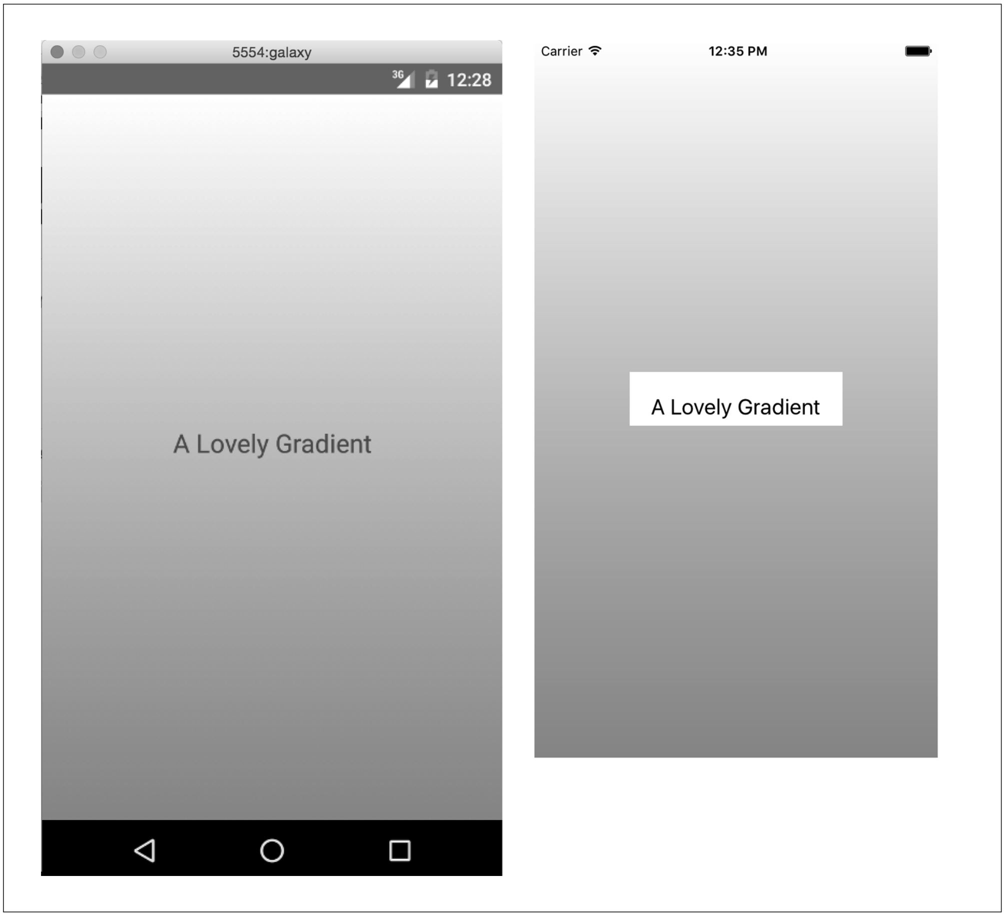


图 7-9：`<Gradient>` 组件，Android（左）和 iOS（右）

创建一个跨平台原生组件是如此便捷，并且不需要太多额外的配置。一旦分别实现了 iOS 和 Android 版本，就只需要创建一个包含 index.ios.js 和 index.android.js 文件的目录即可。每一个版本都需要导入正确的模块。然后你就可以直接引入这个目录，React Native 会自动选择平台对应的版本。

例如，react-native-linear-gradient/ 目录（在 node_modules/ 目录下，它是通过 npm 安装的）下包含一个 index.ios.js 和一个 index.android.js 文件。React Native 将会为我们选择一个正确的文件，我们只需要像这样导入 react-native-linear-gradient/ 即可：

```
var LinearGradient = require('react-native-linear-gradient');
```

按照目前的情况，React Native 不强制在 iOS 和 Android 版本之间保持接口的一致性，因此这是你需要考虑的事情。如果你希望 iOS 和 Android 版本在接口上有一些细微的区别，当然也是可以的！

7.5 小结

何时适合使用原生的 Objective-C 或 Java 代码？何时适合引入第三方模块或类库？总体而言，原生模块有三个使用场景：利用现有的 Objective-C 或 Java 代码；编写像图形处理这样高性能或多线程的代码；暴露 React Native 中未支持的接口。

对于现有的 Objective-C 或 Java 项目来说，编写原生模块是一个在 React Native 中复用现有代码的很好的做法。混合型应用有些超出本书范围，但它确实是一种切实可行的办法，并且你可以使用原生模块在 JavaScript、Objective-C 和 Java 中共享功能代码。

类似地，对于一些注重性能或执行特定任务的场景来说，使用对应平台的原生语言进行开发是明智的做法。在这些场景下，让 Objective-C 或 Java 执行一些繁重的“体力活”，然后把结果传回给 JavaScript 应用的做法更加切实可行。

最终，你难免遇到需要使用但 React Native 尚未支持的平台接口。React Native 还处于开发阶段，因此对特定平台的支持基本将长期处于有待完善的状况。在这种情况下，你有两种解决方案：一种是转向社区，寻找是否有人已经解决了你的问题；另一种则是自己来解决，顺利解决了之后还可以把方案贡献给社区！能够自己编写原生模块，意味着你不需要依赖 React Native 核心团队就可以利用宿主平台了。

即使你之前没有原生 iOS 或 Android 的开发经验，如果你计划使用 React Native 进行开发，尝试阅读 Objective-C 或 Java 代码是一个不错的主意。自己动手尝试并解决问题的能力是无价的，日后在使用 React Native 时万一遇到困难也不会手足无措，并且原生模块的开发难度实际上相当平缓。不用担心，去试试吧！

当你开发自己的 React Native 应用时，React Native 社区以及 JavaScript 完善的生态环境将会给你提供宝贵的资源。你可以在别人的模块的基础上进行开发，如果需要帮助的话，不妨联系他们吧！

第 8 章

调试与开发者工具

当你开发自己的应用时，可能会遇到一些问题。当需要调试应用的时候，我们庆幸地发现已经有一些 React Native 的专用工具了，它们可以帮助我们更轻松地完成工作。React Native 和宿主平台的交互层可能会出现一些糟糕的 bug，本章也会介绍这部分内容。我们还将了解 React Native 开发中的一些常见的陷阱，以及一些用来解决这些问题的工具。不涉及测试内容的调试讲解都是不完整的，因此我们也会介绍一些 React Native 代码自动化测试的内容。

8.1　JavaScript调试实践和解释

当使用 Web 平台上的 React 时，我们拥有大量基于 JavaScript 的通用的技术和工具来帮助我们调试应用。它们中的大多数也适用于 React Native，但有时需要一些微小的改动。React Native 允许我们使用熟悉的控制台、调试器和 React 开发者工具，因此调试 React Native 中的 JavaScript 问题应该会让你感觉很熟悉。

8.1.1　激活开发者选项

为了让自己使用这些工具，你需要在应用内的开发者菜单中启用 Chrome 开发者工具（图 8-1）。这个菜单可以在 iOS 模拟器中通过快捷键 Command+Control+Z 来触发，也可以通过按下 Android 的硬件按钮或摇晃设备来触发。你可以在菜单中选择 Debug in Chrome 来启动 Chrome 开发者工具。

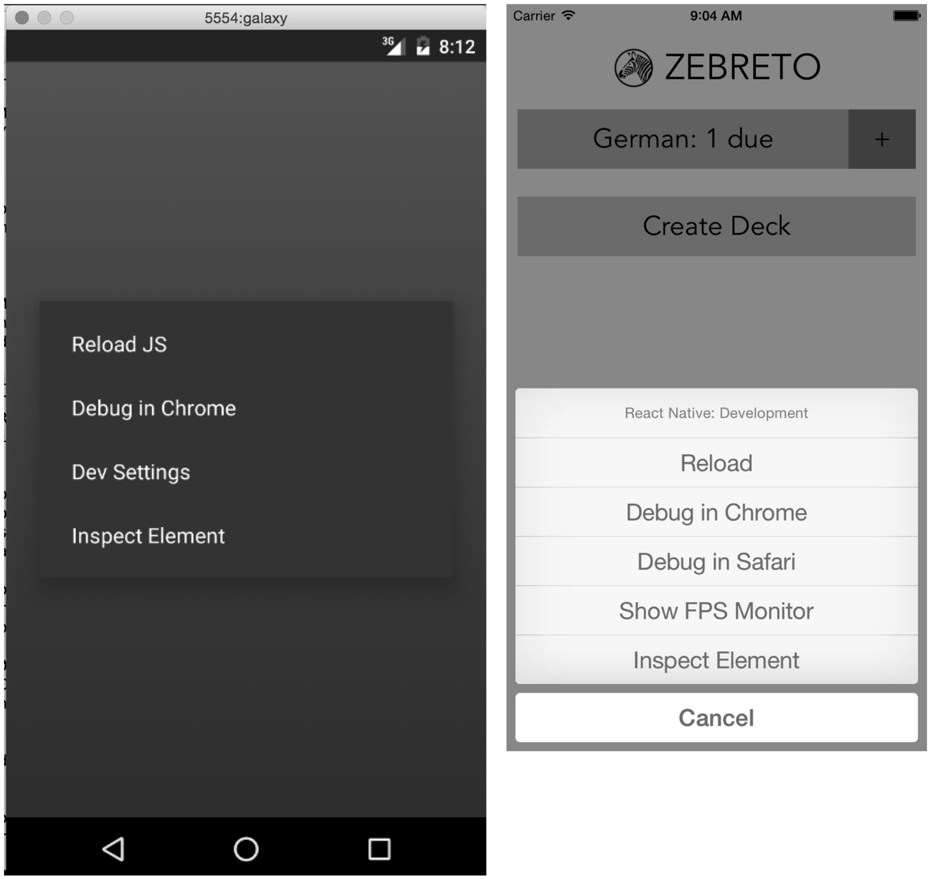


图 8-1：应用内置的开发者菜单，Android（左）和 iOS（右）视图

8.1.2 使用console.log调试

最基础也是最常用的调试方法之一就是先输出，然后观察发生了什么。对于大多数 Web 开发者而言，在代码中添加 `console.log` 是工作流程中的一个潜在环节。

JavaScript 控制台直接在 React Native 之外工作。使用输出语句不需要任何特殊的配置。

使用 Xcode 时，你会看到与 Xcode 控制台相同的输出内容（图 8-2）。注意，你可以通过调整 Xcode 可视面板来扩大控制台的空间。

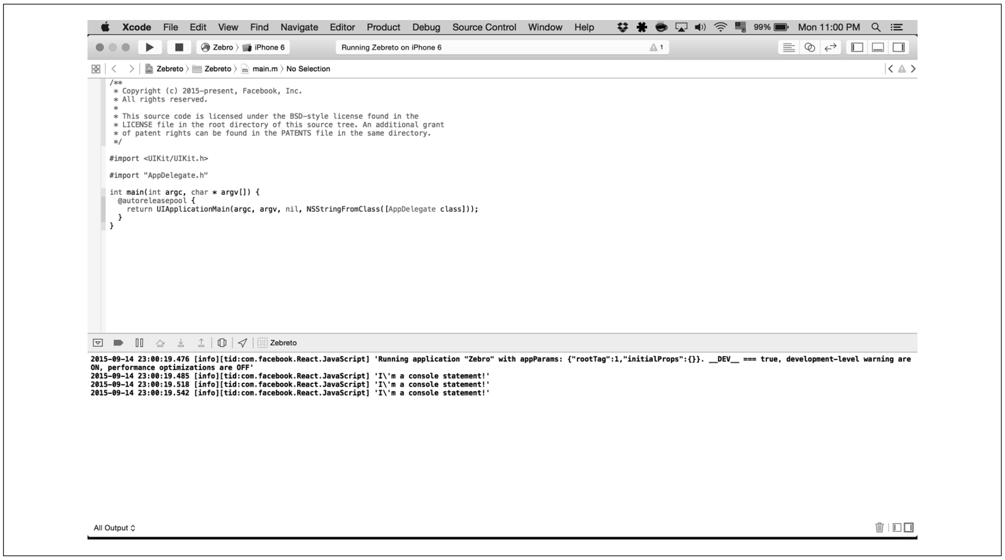


图 8-2：Xcode 中的控制台输出

与此相似，对于 Android 而言，你可以在项目根目录运行 `logcat` 命令来查看设备的日志（图 8-3）。

```
adb logcat
```

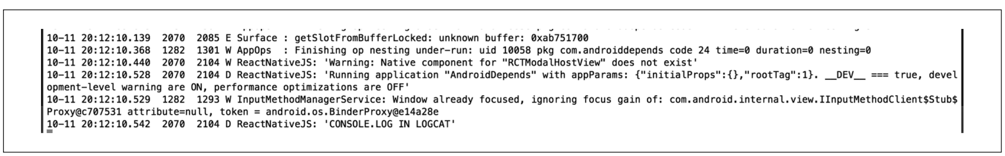


图 8-3：`logcat` 中标签名为“ReactNativeJS”的控制台输出

然而，这些视图杂乱无章，还夹杂着一些平台相关的内容。我们可以直接使用基于浏览器的开发者工具。激活开发者菜单并选择 Debug in Chrome，然后打开你的控制台。如图 8-4 所示，你可以从 Chrome 开发者工具的控制台中看到输出的内容。



图 8-4：Chrome 控制台输出

注意，你需要先打开控制台才可以看到输出内容。

这个功能是怎样工作的呢？当你加载了开启 Chrome 调试工具的 React Native 应用之后，Google 的 Chrome 浏览器便会通过 React Native 包管理器使用一个标准的 `<script>` 标签来执行相同的 JavaScript 代码，因此你可以拥有一个基于浏览器的调试器。随后，包管理器使用 WebSocket 进行设备与浏览器之间的通信。

我们不需要过多关注具体的细节，只需要知道如何使用这些工具就行了！

8.1.3　使用JavaScript调试器

同时，你也可以使用 JavaScript 调试器，就像在开发 Web 平台上的 React 一样。在 Chrome 浏览器打开开发者工具，切换至 source 选项，然后在断点处调试器会被激活。你可以在图 8-5 查看这些步骤。

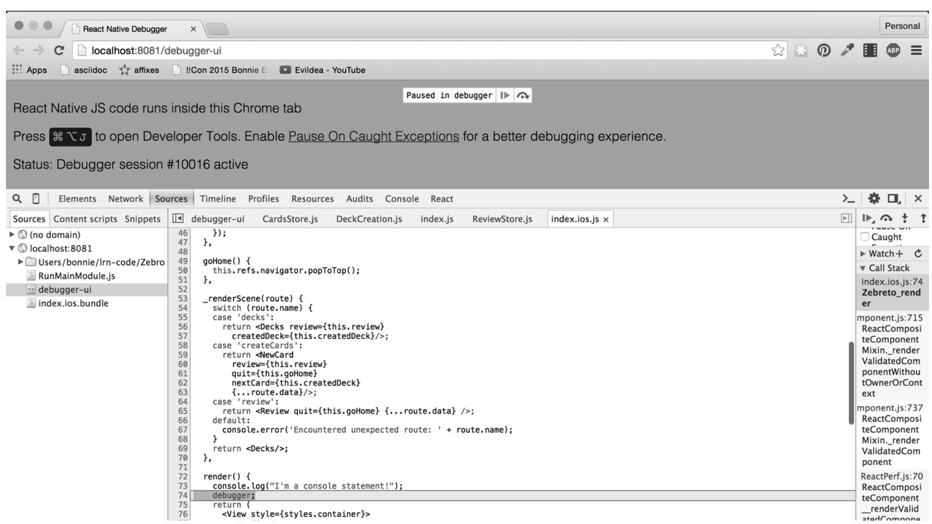


图 8-5：使用调试器

值得一提的是，类似于 JavaScript 控制台，如果没有预先打开开发者工具面板，那么调试器可能不会在断点处被激活。同样，如果没有启用 Chrome 的调试功能，那么调试器也不会被激活。

使用调试器，你就可以直接在 Chrome 上查看跟平时一样形式的源代码。同时，也可以使用浏览器内置的控制台（console）与当前的 JavaScript 上下文进行交互。

8.1.4 使用React开发者工具

开发 Web 平台上的 React 时，React 开发者工具是非常实用的。它允许你审查组件的层次结构，以及检查组件状态和属性，还可以直接在浏览器中修改状态。React 开发者工具可以在 Chrome 扩展中心找到（https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi）。

React Native 也可以使用 React 开发者工具，但体验可能有些不同。打开开发者工具之后，为了让“桥接”生效，暂时需要跟应用进行交互（在屏幕上轻触即可），如图 8-6 和图 8-7 所示。

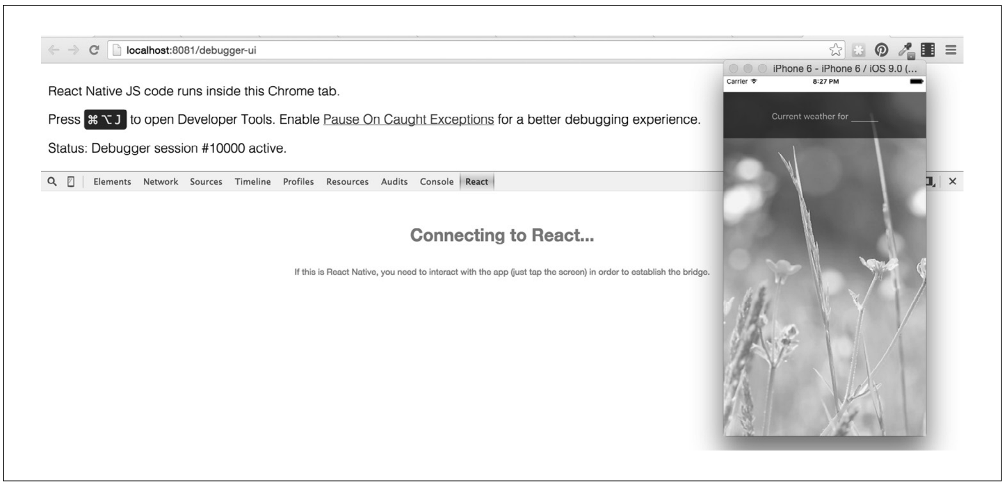


图 8-6：开发者工具启动之前的画面（需要在屏幕上轻触）



图 8-7：使用 React 开发者工具查看组件和属性

这些工具在 React Native 中有一些限制。React Native 专用的开发者工具还处于开发阶段，目前尚未完整支持所有的功能，例如在检查器中修改样式的功能还不太完善。

另一个值得注意的地方是，许多组件缺少 `displayName`。你通常可在 Web 平台的 React 中隐式地设置组件的 `displayName`，就像这样：

```
import React from 'react';
var ComponentName = React.createClass({
  ...
});
export default ComponentName;
```

如果这样定义了一个组件，那么当你使用 React 开发者工具审查元素时，会发现它们显示了对应的名称。由于 React Native 的这一功能有缺陷，你需要显式地设置 `displayName`：

```
import React from 'react-native';
var ComponentName = React.createClass({
  displayName: 'ComponentName'
  ...
});

export default ComponentName;
```

依此设置之后，React 开发者工具就可以正常地解析带有标签的组件层次结构。

8.2 React Native调试工具

除了基于 JavaScript 的 Web 基础调试工具之外，还有一些调试功能是针对 React Native 的。

8.2.1 使用审查元素功能

在浏览器上使用 React 开发者工具时，你会发现“审查元素”功能还有一些待改进的地方。不过应用内的“审查元素”的功能可能会对你有所帮助，它支持查看样式等信息，还为你提供了一种快捷的挖掘组件层级的功能。如图 8-8 所示，你可以查看按钮组件的审查结果。



图 8-8：使用审查元素功能，点击组件查看更多信息

这个视图同时也展示了一些基本的性能指标。

8.2.2　宕机红屏

在应用开发过程中最常见的场景之一就是宕机红屏了。错误界面虽然让人惊慌，但实际上是非常好用的：它捕获错误并解析成有意义的信息。同样，学会理解它显示的错误信息也有助于提高开发效率。

例如，一个语法错误会输出如图 8-9 所示的信息，它指出了错误所在的文件和行号。



图 8-9：语法错误的宕机红屏

另一个常见的错误是尝试使用一个未导入或未定义的变量。例如，不明确地导入 <Text> 组件，像这样：

```
import React from 'react-native';

export default React.createClass({
    render() {
      return (
        <View>
          <Text>
            I haven't required things properly!
```

```
          </Text>
        </View>
      );
    }
  })
```

这会引发如图 8-10 所示的错误信息。



图 8-10：忘记导入 <Text> 组件而引发的错误

尝试使用一个未定义的变量也会引发错误（图 8-11）。



图 8-11：尝试使用未定义的变量的错误信息

样式相关的错误有一些特定的信息。举例而言，如果调用 `StyleSheet.create` 方法并传入了一个无效的值，那么 React Native 将会告诉你哪些值才是有效的（图 8-12）。



图 8-12：误设样式属性的错误信息

宕机红屏看起来可能有点吓人，但确实对你有所帮助，并且它展示的错误信息都是很有用的。如果由于某些原因需要退出错误界面，只需要在模拟器上按下“退出键”（Escape Key）就可以切换回应用界面了。

8.3 JavaScript之外的调试方法

由于使用 React Native 编写移动应用，你遇到的错误可能不仅仅存在于 React 代码里，也可能出现在应用内部。如果你是移动开发的新手，这些问题可能会让人沮丧。此外，有时候你会遇到一些因 JavaScript 代码与宿主平台交互而产生的错综复杂的问题和错误，而

React Native 和宿主平台代码的混合会产生一些莫名其妙的"症状"。

学习调试纯 JavaScript 代码之外的问题对使用 React Native 进行高效率开发来说是非常重要的。值得庆幸的是，很多问题都比想象中容易解决，而且还有大量的工具可以帮助我们。

8.3.1 常见的开发环境问题

React Native 更新得非常快，这意味着管理你的开发环境可能会有点令人懊恼。

如果你遇到由于包管理器的启动，或者由于使用 `npm start` 或 `react-native run-android` 构建或运行应用而产生的错误，那么很可能是依赖的问题。

如果你使用 brew 管理依赖，那么推荐保持 brew 在最新状态：

```
brew update
brew upgrade
```

如果已经升级了 React Native，那么推荐运行这些命令升级你的 node：

```
brew upgrade node
```

此外，你也可以运行 `brew doctor` 来检查已安装的包是否存在问题。

如果你的依赖出现问题，另一种常用的解决办法是清理所有已安装的 npm 包，然后重新安装：

```
rm -rf node_modules
npm install
```

8.3.2 常见的Xcode问题

当你构建 iOS 应用时，可能会遇到一些错误，它们会出现在 Xcode 的问题面板里（图 8-13）。你可以通过选择警告图标来查看错误。

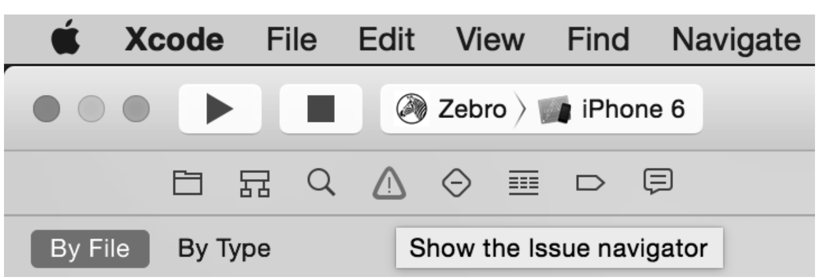


图 8-13：查看错误面板

接着，Xcode 会为你指出相关的文件和行号，在 IDE 里高亮显示这些问题。图 8-14 展示了一个常见错误的例子。

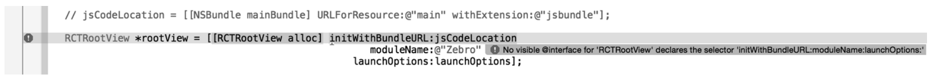


图 8-14：接口错误

No visible interface for“RCTRootView”说明 React Native 的类由于某些原因对 Xcode 来说是不可见的。通常，如果你在 Xcode 里碰到 X is undefined 这样的问题，并且 X 是以 RCT 为前缀或是 React Native 的一部分文件，那么建议你检查包管理器，确保 JavaScript 的依赖处于良好状态中：

(1) 退出包管理器；
(2) 退出 Xcode；
(3) 在项目目录下运行 **`npm install`**；
(4) 重新打开 Xcode；

另一个常见的是资源尺寸的问题（如图 8-15 所示）。

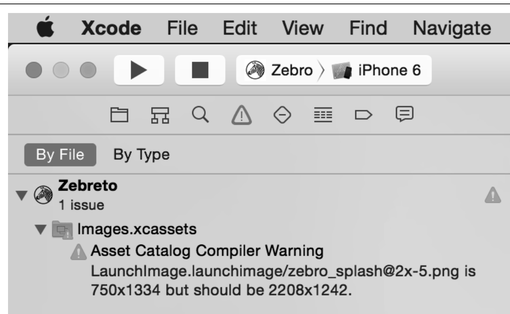


图 8-15：有关图像尺寸问题的警告

资源文件需要符合目标平台的尺寸要求（尤其是应用的图标文件），如果你导入了一个错误的尺寸，那么 Xcode 将会抛出一个警告。

要弄懂 Xcode 的警告可能需要一些时间，尤其当你对 Objective-C 不熟悉时。其中最棘手的是关于 React Native 与 Xcode 项目整合的问题，但通过清理并重新安装 React Native 通常能够得到解决。

8.3.3　常见的Android问题

当你运行 `react-native run-android` 时，可能会出现一些错误，阻止你加载应用。

最常见的两个问题一般是 Android 依赖的丢失，以及未能启动 Android 虚拟设备（或通过 USB 连接的可识别的设备）。

如果你遇到一个关于包丢失的警告，可通过运行 `android` 来查看该包是否被列为“已安装”。如果没有的话，那就安装它。如果已经安装了，但 React Native 无法找到的话，那么可以按照上面的步骤尝试去修复开发环境问题。同时，你也应该检查一下，确保 `ANDROID_HOME` 环境变量已经被正确设置并指向了 Android SDK 的安装目录。例如，在我的系统上有：

```
$ echo $ANDROID_HOME
/usr/local/opt/android-sdk
```

如果你遇到一个关于指定了不可识别设备作为构建目标的问题，那就需要检查你的设备。你是打算在模拟器上运行应用吗？如果是的话，运行 `android avd`，然后启动一个合适的模拟器。如果模拟器仍处在启动过程，`react-native run-android` 命令会执行失败，可以等待几分钟再重试。如果你打算使用物理设备，请确保 USB 调试选项已被激活。

创建签名版本的 Android 应用之后，可能还会遇到一些问题，第 11 章将会讨论这部分内容：

```
$./gradlew installRelease
...
INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES:
New package has a different signature
```

这个问题可以通过在设备或模拟器上卸载旧版本应用来解决，也可以重试安装步骤。它是由于尝试使用一个不同的签名密钥来安装应用而导致的——这当然会在你生成第一个签名版 APK 之后发生。

8.3.4 React Native包管理器

React Native 依赖于包管理器来重新构建代码，因此包管理器的异常会很快以错误的形式展现出来。

当你从 Xcode 或使用 `react-native run-android` 命令运行项目时，React Native 包管理器会自动加载。但在关闭项目的时候，它并不会自动退出。这意味着，如果你切换了项目，那么包管理器仍在运行，只不过运行在错误的目录下，因此代码会编译失败。所以请确保包管理器永远运行在项目对应的根目录下。你可以通过 `npm start` 命令来启动它。

如果 React Native 包管理器在启动时抛出了一些奇怪的错误，那么很可能是你的开发环境处于不良的状态。我们可以根据之前描述的步骤来解决这个问题，确保 npm、node 和 react-native 的本地文件处在良好的状态。

8.3.5 部署至iOS设备的问题

当你尝试在真实的 iOS 设备上测试应用时，可能会遇到一些奇怪的问题。

如果在上传应用到 iOS 设备的过程中遇到麻烦，你要做的第一件事情就是检查是否正确设置了 Apple 开发者账号。打开 iTunes Connect 网站（http://itunesconnect.apple.com/）进行检查，接受所有未决定的协议。如果没有 Apple 开发者账号，你将无法部署应用到设备中。[1]

接下来，确保已经正确选择了你的设备作为构建目标。你的设备是项目设置里支持的类型吗？例如，如果你的应用明确地禁用了 iPad 设备，那么你就不能部署到 iPad 上了。

如果你修改文件时使用 React 包管理器重新构建的话，可能会遇到图 8-16 中的错误。

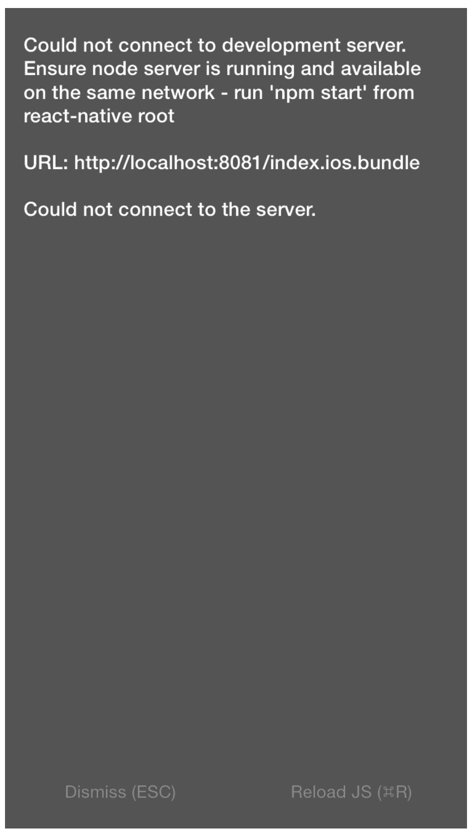


图 8-16：无法连接到开发服务器

注 1：申请 Apple 开发者账号用于真机测试是免费的。——译者注

这表示你的应用已经尝试过从 React Native 包管理器加载 JavaScript 的打包文件，但它无法被成功加载。在这种情况下，试试下列检查步骤。

- 你是否在 AppDelegate.m 里使用了包管理器选项?
- AppDelegate.m 中的 IP 地址是否正确?
- 你的电脑与 iOS 设备是否在相同的 WiFi 网络下?
- React Native 包管理器是否运行在项目目录下?
- 是否在电脑上可以访问 http:// 你的 IP 地址 :8081/index.ios.bundle ?
- 是否可以在 iOS 设备上访问这个网址?

如果你使用了预构建的打包文件，可能会遇到另一个问题，即默认的 `react-native bundle --minify` 命令会把打包的文件放置在错误的路径下。这个问题很容易解决，根据错误信息提示把 main.jsbundle 文件放置到正确的路径下即可。

8.3.6 模拟器行为

你可能会不时发现设备模拟器的一些奇怪的行为。如果你的应用持续不断地崩溃，或者修改后的代码无法在模拟器上反映出来，那么首先试试最简单的办法，即从设备上删除你的应用。

值得注意的是，简单地删除应用可能达不到预期的效果；在很多系统上，被卸载的应用会有一些残留的文件，这些文件在今后可能会有副作用。如图 8-17 所示，最直接的办法就是重置整个设备模拟器，一切从头再来。这样，模拟器上所有的文件和应用都将被移除。



图 8-17：Reset Content and Settings... 选项将会清空设备

8.4　测试代码

学会调试固然很好，但你一定想在错误发生之前就阻止它们（如果不可避免，就捕获它们）。自动化测试和静态类型检查是非常实用的工具，你一定也想在应用里使用它们。

> **测试 JavaScript 代码**
>
> 大部分你写的 React Native 代码可能甚至没有意识到它们被运行在移动环境里了。例如，所有的业务逻辑都可以从渲染逻辑中隔离出来。这意味着你可以使用任何你喜欢的普通 JavaScript 开发工具来测试你的 JavaScript 代码！

这一节将具体介绍如何使用 Flow 进行类型检查，以及如何使用 Jest 进行单元测试。

8.4.1　使用Flow进行类型检查

Flow（http://flowtype.org/）是一个静态类型检查的 JavaScript 类库。它依赖类型推断来检测类型错误，甚至也可以检查注释代码。它允许你逐渐往现有的项目里添加注解。类型检查可以帮助你尽早发现潜在问题，然后增强不同组件和模块之间的 API 的健壮性。

运行 Flow 是很简单的：

```
$ flow check
```

应用默认自带了 .flowconfig 文件，它配置了 Flow 的行为。如果你发现了很多关于 `node_modules` 的错误，可能需要添加这一行到 .flowconfig 文件的 `[ignore]` 下面：

```
.*/node_modules/.*
```

再次运行 `flow check`，就没有任何错误了：

```
$ flow check
$ Found 0 errors.
```

尽情使用 Flow 来帮助你开发 React Native 应用吧。

8.4.2　使用Jest进行测试

React Native 支持使用 Jest 来测试 React 组件。Jest 是一个基于 Jasmine 的单元测试框架。它提供了侵入性的依赖自动模拟的功能，也可以很好地与 React 测试工具进行整合。

使用 Jest 需要先进行安装：

```
npm install jest-cli --save-dev
```

更新 package.json 文件，在 `scripts` 中添加 `test`：

```
{
  ...
  "scripts": {
    "test": "jest"
  }
  ...
}
```

运行 `npm test` 命令之后，将会启动 `jest`。

接下来，创建一个 tests/ 目录。Jest 将会递归地搜索在 tests/ 目录下的测试文件，然后运行它们：

```
mkdir __tests__
```

现在创建一个新文件 tests/dummy-test.js，并编写我们的第一个测试用例：

```
'use strict';

describe('a silly test', function() {
 it('expects true to be true', function() {
   expect(true).toBe(true);
 });
});
```

现在，如果你运行 **`npm test`**，会看到测试用例全部通过了。

当然，除了这个简单的例子之外，测试还包含更丰富的内容。你可以在 React Native 仓库中的 Movie 示例应用中获得更多参考。

举例来说，这里有一个缩减版的 Movie 示例应用中 `getImageSource` 功能的测试文件（GitHub 上有完整的代码可用：https://github.com/facebook/react-native/blob/0.12-stable/Examples/Movies/__tests__/getImageSource-test.js）。

```
jest.dontMock('../getImageSource');
var getImageSource = require('../getImageSource');

describe('getImageSource', () => {
  it('returns null for invalid input', () => {
    expect(getImageSource().uri).toBe(null);
  });
  ...
});
```

需要注意的是，你需要先显式地阻止 Jest 模拟文件，然后再引入你的依赖文件。如果想了解更多关于 Jest 的信息，建议从它的文档开始（https://facebook.github.io/jest/）。

8.5　当你陷入困境

如果你遇到一个特别棘手的问题，并且自己无法解决的话，可以尝试咨询社区。你可以去这些地方寻求建议：

- #reactnative IRC 聊天室（irc://chat.freenode.net/reactnative）
- React 讨论论坛（https://discuss.reactjs.org/）
- Stack Overflow（http://stackoverflow.com/questions/tagged/react-native）

如果怀疑所遇到的问题可能是 React Native 自身的 bug，可以去 GitHub 检查现有的问题列表（https://github.com/facebook/react-native/issues）。在提交问题的时候，使用一个小的示例程序帮助你说明问题通常是很实用的。

8.6　小结

总体来说，调试 React Native 与调试 Web 平台上的 React 应该会有非常相似的体验。大多数你熟悉的工具在这里依然可用，这可以让我们更容易地过渡到 React Native 开发。话虽如此，React Native 应用有它特有的复杂性，有时候这种复杂性会表现为一些令人沮丧的 bug。了解调试应用的方法以及环境产生的错误信息，将会对你形成高效的工作方式有持续的帮助。

第 9 章

学以致用

前文中已经介绍了许多构建 React Native 应用的知识，本章将整合并应用这些知识。目前所介绍的大部分是小型应用，本章将介绍大型应用的基本结构，以及 Reflux（一个基于 Flux 模型的单向数据流类库）的使用。本章还会介绍如何使用 Dimensions 接口进行不同尺寸屏幕的适配。本章最后布置了一些课后任务：在现有的 React Native 代码基础上开发更多的功能，看看感觉如何。

9.1 闪卡应用

Zebreto 是一个闪卡应用，它的产生是基于间隔重复学习法（Spaced Repetition System，SRS），一种高效记忆的学习策略。间隔重复学习法的目标是在你忘记之前就让你复习需要记忆的知识。如果你学过外语，也许会熟悉间隔重复学习法。这种方法可以让你更快地记住大量信息，以达到永久记忆的目标。一个常用的办法是在较短间隔时间内进行复习，比如一个小时，然后随着正确率的提高逐渐延长间隔：先是一个小时，然后延长到一天，接着延长到三天，最后延长到一个星期。记忆间隔可以逐渐延长到一年或五年之久。使用传统的纸笔和卡片来使用这种学习法是不切实际的，因此我们开发了这个应用。

Zebreto 会比我们之前开发的示例应用更复杂一些。学习它主要是为了了解更复杂的应用应该怎样开发。该应用的所有代码都在 GitHub 上（https://github.com/bonniee/learning-react-native/tree/master/Zebreto），而且是完全跨平台的，可以运行在 Android 平台上并与 iOS 上的表现相一致。

如图 9-1 所示，Zebreto 应用有三个主要的视图。

- 主界面，列出了存在的分组，并且可以创建新的分组。
- 创建卡片界面。
- 复习界面。

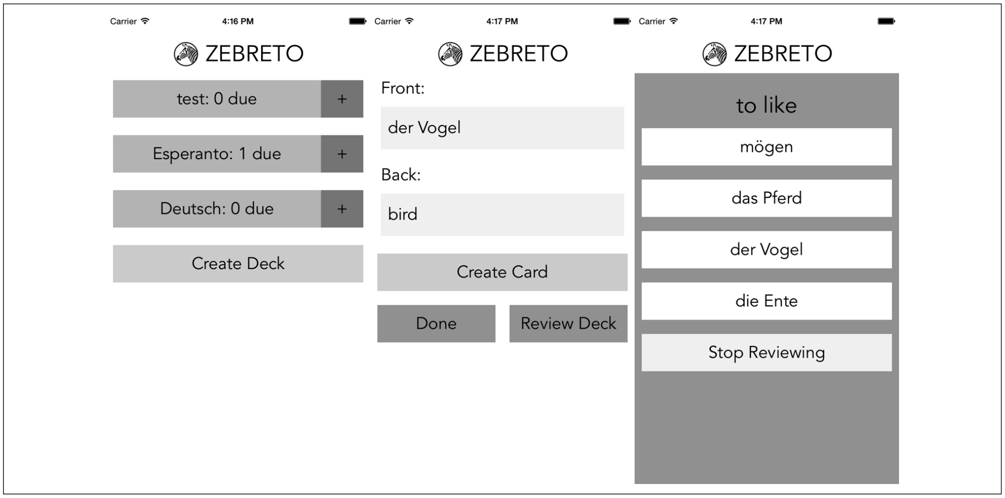


图 9-1：查看分组、创建卡片以及复习卡片

该应用的用户主要有两个交互流程。一个是内容的创建（即创建分组和卡片）。创建内容的流程如下（如图 9-2 所示）。

(1) 用户点击 Create Deck 按钮。
(2) 用户输出一个分组名称，然后轻触返回按钮或点击 Create Deck 按钮继续创建。
(3) 用户在 Front 和 Back 输入框输入内容，然后点击 Create Card。
(4) 输出零个或多个卡片之后，用户可能会点击 Done 按钮返回初始界面，或者可以点击 Review Deck 按钮进行复习。

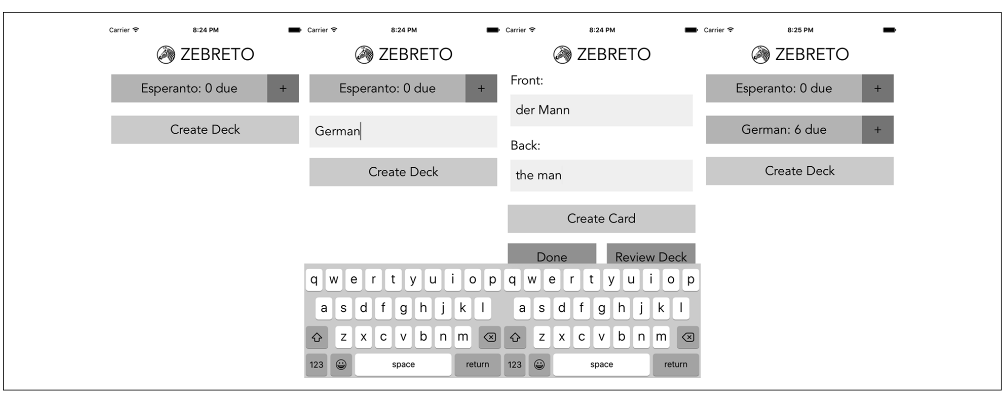


图 9-2：创建一个分组

通过点击主界面上的“+”按钮，还可以在今后继续创建卡片。

第二个主要的交互流程是卡片的复习（如图 9-3 所示）。

(1) 用户点击需要复习的分组名。
(2) 展现问答界面给用户。
(3) 用户点击其中一个选项。
(4) 应用根据猜测正确与否来反馈给用户。
(5) 点击 Continue，继续复习。
(6) 所有的复习完成之后，用户会看到 Reviews cleared! 界面。



图 9-3：复习卡片

如果用户猜对了卡片，那么我们要增加卡片的熟练值，并且推迟下次卡片出现的时间。同样，对于每一个猜错的卡片，我们要减少卡片的熟练值，并且尽快再安排一个复习任务。

我们将使用 Zebreto 应用，尤其是上面描述的特性，来讨论一些关于构建完整应用的模式以及出现的问题。

9.1.1 项目结构

这里是项目大体的结构：

```
Zebreto
 |- .babelrc
 |- iOS
 |- index.ios.js
 |- node_modules
 |- package.json
 |- src
    |- actions.js
    |- components
    |- data
```

```
        |- stores
        |- styles
```

在 Zebreto 目录内，我们的项目简单划分了 iOS 和 Android 以及 src/ 子目录。src/ 目录包含项目中所有的 React 代码。同时注意 .babelrc 文件，它修改了 Babel 默认的配置。如果你在 React Native 项目的根目录添加一个 .babelrc 文件，React Native 包管理器将会自动加载它。在这种情况下，最大的改变是我们启用了 ES6 模块语法。

```
// .babelrc

{
  "stage": 1,
  "optional": ["runtime"],
  "loose": "all",
  "whitelist": [
    "es6.modules"
  ]
}
```

大部分情况下，我们在 src/ 目录内进行开发工作。

在 src/ 目录内，我们的代码通过功能再进行组织。

- components/
  所有的 React 组件都在这里

- data/
  你可以在这里找到数据模型

- stores/
  我们的 Reflux 的数据 store（存储），很快我们会讨论到它

- actions.js
  Reflux 的 action（动作），我们将与数据 store 一起讨论

- styles/
  这里可以找到样式对象，支持在任何地方复用

9.1.2　组件层次结构

该应用有三个场景，它们可以在任何时候被展现。为了让你对应用的总体结构有一些认识，我们画出这三个场景的组件树。

首先，我们可以在主界面创建分组。这个界面将会显示现有的所有分组，如图 9-4 所示。组件层次结构如图 9-5 所示。



图 9-4：从主界面创建分组

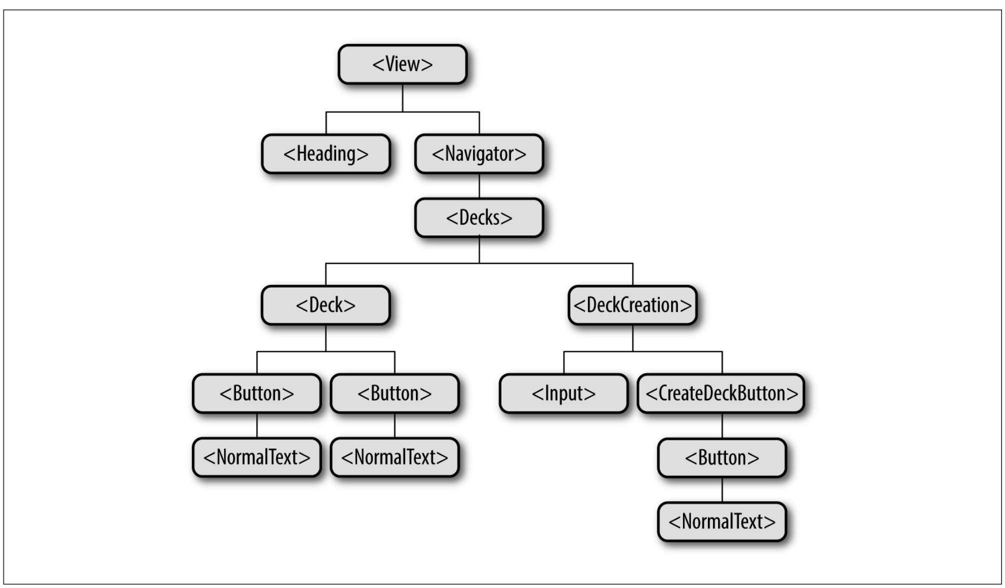


图 9-5：分组创建界面的组件树

紧接着是卡片创建界面（图 9-6）。



图 9-6：卡片创建界面

该界面的组件层次结构如图 9-7 所示。

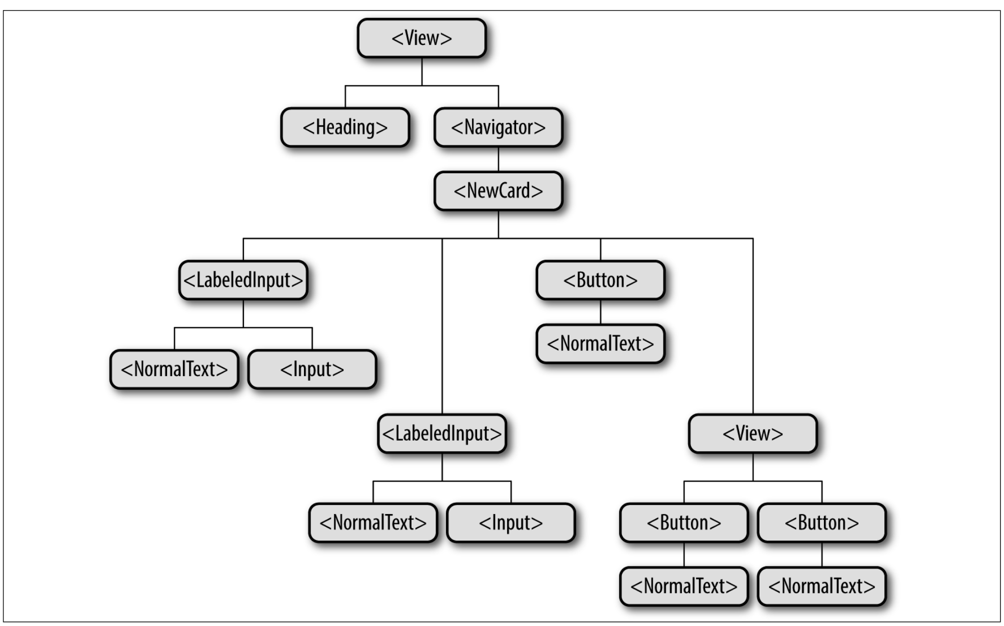


图 9-7：分组创建的组件树

最后，我们有复习界面，如图 9-8 所示。你也许会注意到，`<Review>` 组件的子组件会根据不同的复习流程而发生改变。用户完成所有复习之后，`<ViewCard>` 组件将会被替换成用户复习表现的反馈信息。



图 9-8：卡片复习界面

卡片复习界面的组件层次结构如图 9-9 所示。

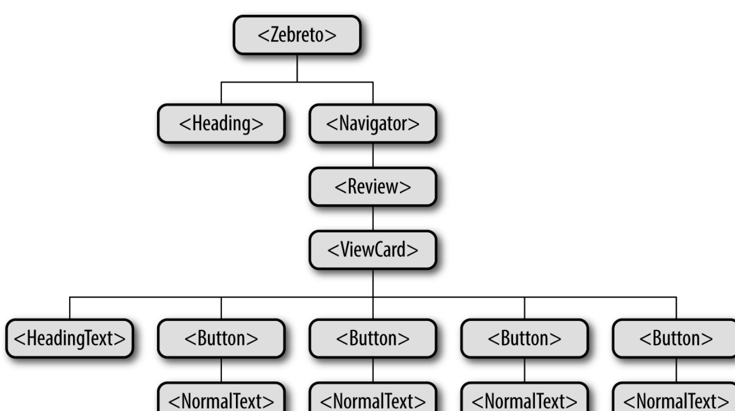


图 9-9：分组创建界面的组件树

如前所述，当你开发大型应用时，编写能不断复用的样式组件是非常有用的。如此一来，大多数的组件都不需要使用 `<Text>` 组件来渲染，而是使用 `<HeadingText>` 和 `<NormalText>` 组件来代替。类似地，`<Button>` 组件也经常会被复用。这样做可以提高代码的可读性，让创建新组件变得更加容易。

以上内容应该会让你对 Zebreto 应用的结构有一些感性认识。目前为止，我们尚未讨论用户交互以及数据的修改与持久化存储这样的内容。我们先来看数据模型。

9.2 模型与数据存储

既然我们已经了解 Zebreto 是怎样处理渲染逻辑的，那么它又是如何处理数据的呢？我们需要记录哪些数据呢？应该要怎么做？

Zebreto 由两个基本模型构成：卡片（Card）和分组（Deck）。

分组包含一个人类可读的名称和一个唯一的 ID。我们有时候也储存一些关于它所包含的卡片的元数据。

```
Deck: {
  name,
  id,
  totalCards, // computed, may be out of date
  dueCards // computed, may be out of date
}
```

卡片有正面（`front`）有反面（`back`）（比如“der Hund”和“the dog”），以及所属的分组（`deckID`）。此外，还有一个表示成整数的熟练值（`strength`），练习的日期（`dueDate`）。Zebreto 使用 moment.js 来处理日期对象：

```
Card: {
  front,
  back,
  deckID,
  strength,
  dueDate,
  id
}
```

分组和卡片应该用简单的 JavaScript 对象来表示，但是为了方便，Zebreto 使用了一些包装类。如果查看 src/data/ 目录，你会找到我们的模型类。这里是一个 Deck 类：

```
// src/data/Deck.js
import md5 from 'md5';

class Deck {
  constructor(name) {
```

```
    this.name = name;
    this.totalCards = 0;
    this.dueCards = 0;
    this.id = md5(name);
  }

  setFromObject(ob) {
    this.name = ob.name;
    this.totalCards = ob.totalCards;
    this.dueCards = ob.dueCards;
    this.id = ob.id;
  }

  resetCounts() {
    this.totalCards = 0;
    this.dueCards = 0;
  }

  static fromObject(ob) {
    let d = new Deck(ob.name);
    d.setFromObject(ob);
    return d;
  }
}

module.exports = Deck;
```

正如你看到的一样，Deck（分组）类非常简单。它的构造函数需要一个区分不同 Deck 的参数，然后为其他属性设置合理的默认值。它也为我们提供了一个从 JavaScript 对象中创建 Deck 的便利的方法，以及一个重置 Deck 元数据的简单的办法。

目前，所谓唯一的 ID 是通过对相关信息做 MD5 散列生成的。

Card（卡片）类看起来非常类似，它为我们提供了一个从普通对象中创建 Card 的助手方法（helper method）：

```
// src/data/Card.js

import md5 from 'md5';
import moment from 'moment';

class Card {
  constructor(front, back, deckID) {
    this.front = front;
    this.back = back;
    this.deckID = deckID;
    this.strength = 0;
    this.dueDate = moment();
    this.id = md5(front + back + deckID);
  }

  setFromObject(ob) {
    this.front = ob.front;
```

```
    this.back = ob.back;
    this.deckID = ob.deckID;
    this.strength = ob.strength;
    this.dueDate = moment(ob.dueDate);
    this.id = ob.id;
  }

  static fromObject(ob) {
    let c = new Card(ob.front, ob.back, ob.deckID);
    c.setFromObject(ob);
    return c;
  }
}

module.exports = Card;
```

为了理解如何在应用中使用这些模型，我们来看一看数据流架构。

9.2.1 数据流架构：Reflux与Flux

Zebreto 使用基于 Flux 模式的 Reflux 来实现数据流架构。本书先前学习的例子都不太需要数据流的管理。在较小的应用中，组件之间的通信通常都不是重要的问题。考虑一下点击按钮会影响父组件的状态的情况。

```
// Child.js
import React from 'react-native';
var {Text, TouchableOpacity} = React;

export default React.createClass({
  render() {
    <TouchableOpacity onPress={this.props.onPress}>
      <Text>Child Component</Text>
    </TouchableOpacity>
  }
});
```

从父组件传递一个回调函数给子组件，父组件可以接收到子组件的交互行为。

```
// Parent.js
import React from 'react-native';
import Child from './Child';

export default React.createClass({
  getInitialState() {
    return {
      numTaps: 0
    }
  },
  _handlePress() {
    this.setState({numTaps: this.state.numTaps + 1});
  },
  render() {
```

```
    <Child onPress={this._handlePress}/>
  }
});
```

对于简单的用例，这个模式可以正常工作。

当我们需要更复杂的交互时，显然我们需要更健壮的数据流架构。当组件树中更低层的组件需要影响高层组件的状态时，应该怎么做呢？让我们再来看看这个界面（图 9-10）。



图 9-10：复习卡片

在你选择一个答案之后，会发生下列动作。

(1) 应用提供视觉反馈，告知选择正确与否。
(2) 下一个卡片准备就绪。
(3) 在适当的时候，更新卡片的熟练值。
(4) 在适当的时候，更新分组中的待复习量。

如果想在复习中途退出 Zebreto 应用时保存当前数据，那么每次选择一个答案后，所有改变的状态都应该被保存起来（图 9-11）。

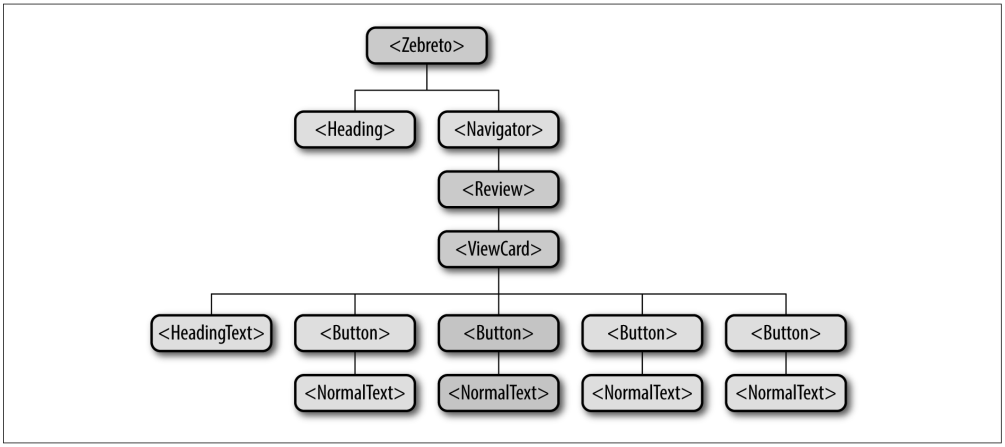


图 9-11：关于复习界面需要了解的组件

顶层的 Zebreto 组件以及 `<Review>` 和 `<ViewCard>` 组件都会接收到这些更新。在这个用例下，传递回调函数的方式无法良好地进行伸缩，因此我们使用类 Flux 的数据架构来完成这项工作。

Flux 更像是一种模式，而不是一个正式的框架，它主要的概念是单向数据流。在 React 中，属性和状态都通过父组件传递给子组件；单向数据流意味着高性能的渲染过程，我们的应用状态也会更容易把握。

传递大量回调函数破坏了清晰的流程，本质上形成了双向数据绑定，可能会出乎意料地触发连锁更新。通过使用类 Flux 应用架构（如图 9-12），我们从可触发的 UI 组件中分离出改变应用状态的逻辑，维持单向模式。

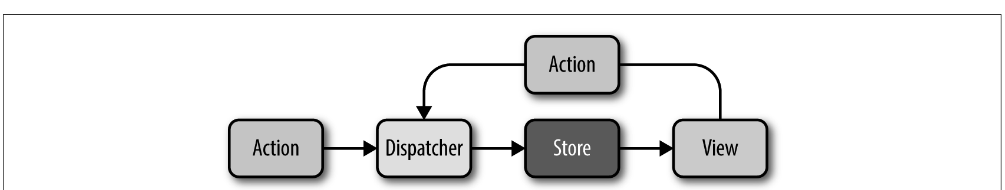


图 9-12：Flux 架构中的数据传播

使用 Flux 模式，视图会根据 Store 中的信息进行渲染。Action 可以由用户与视图的交互或其他事件触发，比如应用的初始化事件。Dispatcher 处理传入的 Action 并将它们传到 Store 中。

Flux 是 Facebook 公司为解决此类问题而提出的官方架构，但 React 社区也有其他一些受 Flux 启发而开发的类库，它们都试图解决相同的问题。Reflux，如图 9-13 所示，就是一个特别出名的类库，我们将会在 Zebreto 里使用它。

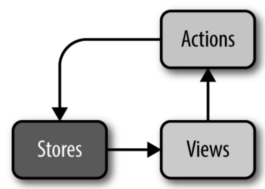


图 9-13：Reflux 架构中的数据传播

Reflux 中没有 Dispatcher 的概念，只有 View、Store 和 Action，其中 Store 可以直接监听 Action。

使用 `npm install` 命令很容易就可以将 Reflux 添加到 React Native 项目中：

```
npm install --save reflux
```

9.2.2　在Zebreto中使用Reflux

我们一起来看看在应用中怎样使用 Reflux。

在 Zebreto 中，我们有多个 Store（如图 9-14 所示）。

- `DeckMetaStore`
  包含分组元数据，比如待复习的数量。
- `CardsStore`
  包含所有卡片。
- `ReviewStore`
  包含当前分组中所有的复习内容。

Store 可以互相监听。Zebreto 中使用从 `CardsStore` 和 `DeckMetaStore` 中获得的信息来创建复习的内容，因此 `ReviewStore` 监听了其他两个 Store。它们之间的关系如图 9-14 所示。

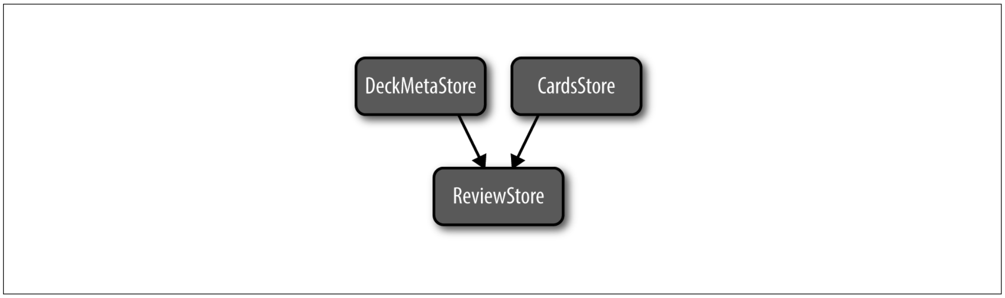


图 9-14：Zebreto 中使用的 Store

actions.js 文件中还有多个 Action:

```
// src/actions.js

import Reflux from 'reflux';

export var DeckActions = Reflux.createActions([
  'createDeck',
  'deleteDeck',
  'reviewDeck',
  'deleteAllDecks'
]);

export var CardActions = Reflux.createActions([
  'createCard',
  'deleteCard',
  'review',
  'editCard',
  'deleteAllCards',
]);
```

任何组件都可以触发被 Store 监听的 Action，因此可能会产生连锁效应。

回到我们复习卡片的例子中，Reflux 数据流工作流程如下（图 9-15）。

- 用户选择一个答案，触发 `CardActions.review` 动作。
- `ReviewStore` 监听了 `CardActions.review` 动作，并处理新的信息。
- `ReviewStore` 在适当的时候触发 `CardActions.editCard` 动作。
- `CardsStore` 监听 `CardActions.editCard` 动作，它会把相关的改变储存到 `AsyncStorage` 中，并触发更新。
- 顶层的 `<Zebreto>` 组件监听 `CardsStore` 中的更新，并相应地更新自身的状态。

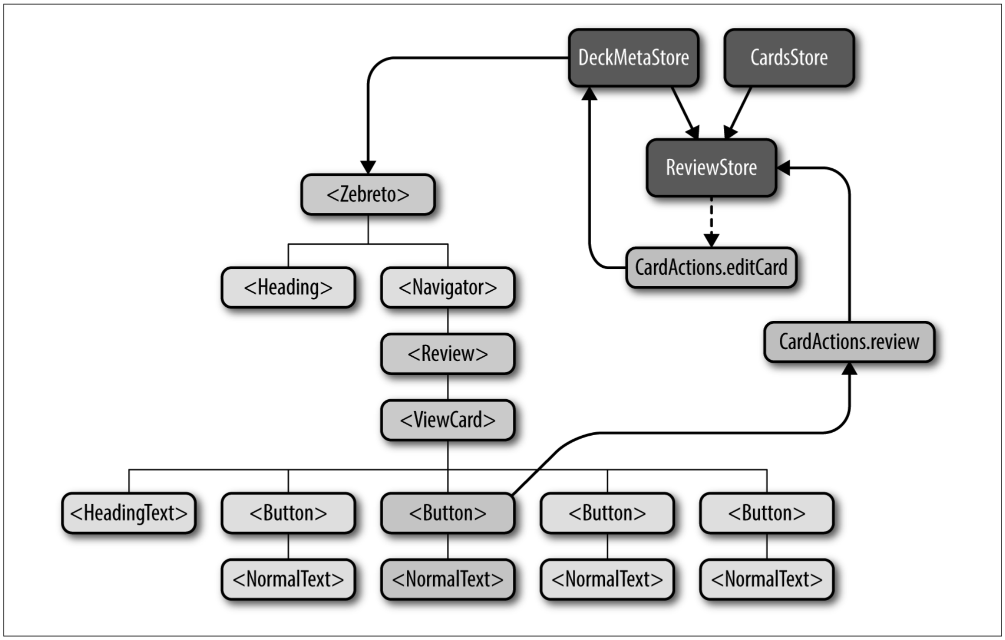


图 9-15：处理卡片复习之后的更新动作

另一个例子是分组的创建，流程如下。

- 创建分组的按钮可以触发一个 `DeckActions.createDeck` 动作。
- 创建分组按钮也可以触发一个属性中的回调函数，让 Navigator 过渡到创建卡片的界面。
- `DeckMetaStore` 监听 `DeckActions.createDeck` 动作；它创建一个新的 Deck 并存储到 `AsyncStorage` 中。

9.2.3 AsyncStorage与Reflux Store的持久化

Zebreto 通过简单的 JSON 序列化将数据存储到 `AsyncStorage` 中。这个过程通过 Store 进行处理，因为 Store 是应用状态实际情况的中心来源。例如，我们看看 `CardsStore` 这个例子。

```
// src/stores/CardsStore.js

import Card from './../data/Card';
import Reflux from 'reflux';
import _ from 'lodash';
import {CardActions} from './../actions';

import React from 'react-native';
var { AsyncStorage } = React;

const CARD_KEY = 'zebreto-cards';
```

```
var cardsStore = Reflux.createStore({
  init() {
    this._loadCards().done();
    this.listenTo(CardActions.createCard, this.createCard);
    this.listenTo(CardActions.deleteAllCards, this.deleteAllCards);
    this.listenTo(CardActions.editCard, this.editCard);
    this._cards = [];
    this.emit();
  },

  async _loadCards() {
    try {
      var val = await AsyncStorage.getItem(CARD_KEY);
      if (val !== null) {
        this._cards = JSON.parse(val).map((cardObj) => {
          return Card.fromObject(cardObj);
        });
        this.emit();
      }
      else {
        console.info(`${CARD_KEY} not found on disk.`);
      }
    }
    catch (error) {
      console.error('AsyncStorage error: ', error.message);
    }
  },

  async _writeCards() {
    try {
      await AsyncStorage.setItem(CARD_KEY, JSON.stringify(this._cards));
    }
    catch (error) {
      console.error('AsyncStorage error: ', error.message);
    }
  },

  deleteAllCards() {
    this._cards = [];
    this.emit();

  },

  editCard(newCard) {
    // 假设newCard.id对应一个存在的卡片。
    let match = _.find(this._cards, (card) => {
      return card.id === newCard.id;
    });
    match.setFromObject(newCard);
    this.emit();

  },

  createCard(front, back, deckID) {
```

```
    this._cards.push(new Card(front, back, deckID));
    this.emit();
  },

  emit() {
    this._writeCards().done();
    this.trigger(this._cards);
  }
});

export default cardsStore;
```

CardsStore 是 Zebreto 应用中唯一处理卡片相关数据读写的地方。现在，我们通过 _loadCards() 和 _writeCards() 函数调用 AsyncStorage 来完成这项工作。如果我们想更换储存卡片数据的方式，例如，更换成 SQLite 数据库或通过网络调用获取数据，那我们只需要更新这两个方法就可以轻松完成。

另一个值得注意的地方是，CardsStore 中的 init() 方法在初始化时会加载储存的数据；任何时候卡片有更新，都会通过 emit() 方法重新存储到 AsyncStorage 中。这样一来，用户无论什么时候退出应用，他们的数据都会被保存起来。

9.3 使用Navigator

Zebreto 中的另一个可能比较有趣的地方是 <Navigator> 组件的用法。我们来看看根组件的源代码：

```
// src/components/Zebreto.js
import React from 'react-native';
var {
  StyleSheet,
  View,
  Navigator
} = React;

import Reflux from 'reflux';
import {DeckActions} from './../actions';

import Decks from './Decks';
import Review from './Review';
import NewCard from './NewCard';
import Heading from './Header';

import CardsStore from './../stores/CardsStore';
import DeckMetaStore from './../stores/DeckMetaStore';

var Zebreto = React.createClass({
  displayName: 'Zebreto',

  mixins: [Reflux.connect(DeckMetaStore, 'deckMetas')],
```

```
componentWillMount() {
  CardsStore.emit();
},

review(deckID) {
  DeckActions.reviewDeck(deckID);
  this.refs.navigator.push({
    name: 'review',
    data: {
      deckID: deckID
    }
  });
},

createdDeck(deck) {
  this.refs.navigator.push({
    name: 'createCards',
    data: {
      deck: deck
    }
  });
},

goHome() {
  this.refs.navigator.popToTop();
},

_renderScene(route) { ❷
  switch (route.name) {
  case 'decks':
    return <Decks review={this.review}
      createdDeck={this.createdDeck}/>;
  case 'createCards':
    return <NewCard
      review={this.review}
      quit={this.goHome}
      nextCard={this.createdDeck}
      {...route.data}/>;
  case 'review': ❸
    return <Review quit={this.goHome} {...route.data}/>;
  default:
    console.error('Encountered unexpected route: ' + route.name);
  }
  return <Decks/>;
},

render() { ❶
  return (
    <View style={styles.container}>
      <Heading/>
      <Navigator
        ref='navigator'
        initialRoute=
        renderScene={this._renderScene}/>
```

```
      </View>
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 30
  }
});

export default Zebreto;
```

这个文件内容较多，我们把它分成几个部分。

❶ render 方法实际上非常简短。我们把所有东西包装在 <View> 组件内，然后渲染包含 logo 的头部；还有 <Navigator> 组件，它渲染适当的场景。正如我们之前所讨论的，我们可以从 _renderScene() 方法中看到三个可能的场景：decks、createCards 和 review。这样，应用顶层由一个包装组件和两个子组件组成。

❷ _renderScene() 同时也要将恰当的数据和回调函数作为属性附加到到每一个场景组件中。这里使用了更方便的传播语法。你之前可能不经常看到传播语法，其实这是取自 ES6 的一个优雅的特性。举例而言，我们像下面的例子这样来调用 _renderScene() 方法，它会返回下一个标注那样的代码。

```
_renderScene({
  data: {
    someProp: 'whatever',
     anotherProp: 2
  }
});
```

❸ 通过传播语法，_renderScene() 方法会返回与下面等效的代码：

```
return (
  <Review
    quit={this.goHome}
    someProp="whatever"
    anotherProp={2} />);
```

好了，这就是我们的 <Zebreto> 根组件，它保留了一个到 <Navigator> 的引用（ref），并且管理了不同的场景。但是大多数情况下，都是在各自场景里实现更复杂的功能。

我们把 <Navigator> 组件和 _renderScene() 逻辑放在顶层组件中，并把 goHome() 这样的回调函数作为属性传到每个场景中去，这样，场景本身就不需要关心导航的结构了。但在这里，我们把所有导航相关的渲染逻辑都放在了 <Zebreto> 组件里。

假如想把 <Navigator> 替换成一些平台特定的组件（例如 <NavigatorIOS> 组件），就会变得非常容易，因为它的用法都在这个文件里（只需分别创建 Zebreto.ios.js 和 Zebreto.android.

js 这两个文件即可）。即使我们现在不需要这么做，让导航更清晰并从顶层组件中隔离出来也是明智的做法。

9.4 探索第三方依赖

我们也来看看应用中使用的外部类库。Zebreto 没有太多的第三方依赖，但仍然会用到一些。看一下 package.json 文件：

```
// package.json

{
  "name": "Zebreto",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node_modules/react-native/packager/packager.sh"
  },
  "dependencies": {
    "lodash": "^3.10.1",
    "md5": "^2.0.0",
    "moment": "^2.10.6",
    "react": "^0.13.3",
    "react-native": "^0.11.2",
    "reflux": "^0.2.12"
  }
}
```

`react-native` 和 `react` 很显然是框架本身，我们也已经对 `reflux` 有所了解。`moment` 用来处理日期对象，`md5` 用来计算卡片和分组的 ID。最后，`lodash` 提供了一些实用的工具方法，我们用它来打乱卡片。

这些类库都不是 React Native 或移动设备自带的，但并不要紧。它们不需要任何修改就可以直接运行了！

9.5 响应式设计与字体尺寸

为了让你的应用良好地支持多种设备，你的界面需要适应各种不同尺寸的屏幕。在某种程度上说，基于 flexbox 的样式帮你解决了这个问题，并且不需要特意留心这些问题。

但是，字体样式通常需要根据屏幕尺寸精确地进行调整。为了适应不同设备的屏幕尺寸，Zebreto 中的可复用文本组件会根据屏幕宽度进行自动缩放（图 9-16）。



图 9-16：iPhone 4s 和 iPhone 6 上字体尺寸有略微的区别

处理不同的字体尺寸相当容易，来看看 Zebreto 是怎样缩放字体的吧。

在 styles/ 目录下，我们从 fonts.js 文件中导出了字体相关的样式表，以及今后会用到的缩放系数：

```
// src/styles/fonts.js
import { StyleSheet } from 'react-native';

var fonts = StyleSheet.create({
  normal: {
    fontSize: 24,
    fontFamily: 'Avenir Medium'
  },

  alternate: {
    fontSize: 50,
    fontFamily: 'Avenir Heavy',
    color: '#FFFFFF'
  },
```

```
  big: {
    fontSize: 32,
    alignSelf: 'center',
    fontFamily: 'Avenir Medium'
  }
});

var scalingFactors = {
  normal: 15,
  big: 7
};

module.exports = {fonts, scalingFactors};
```

接着，在字体组件里，比如 `<NormalText>` 组件，我们获取了屏幕的尺寸（dimension）。引入之后，就可以使用 `Dimensions` 接口了：

```
import Dimensions from 'Dimensions';
let {width, height} = Dimensions.get('window');
```

现在获得了屏幕尺寸，参数存储在 `width` 和 `height` 变量中。

在 `<NormalText>` 组件中，我们只使用了 `width` 值来结合缩放系数以决定字体尺寸：

```
var scaled = StyleSheet.create({
  normal: {
    fontSize: width / scalingFactors.normal
  }
});
```

然后，在组件里使用这个样式表：

```
// src/components/NormalText.js

import React from 'react-native';
var {
  StyleSheet,
  Text,
  View
} = React;

import {fonts, scalingFactors} from './../styles/fonts';
import Dimensions from 'Dimensions';
let {width} = Dimensions.get('window');

var NormalText = React.createClass({
  displayName: 'NormalText',

  propTypes: {
    style: View.propTypes.style
  },

  render() {
```

```
    return (
      <Text style={[this.props.style, fonts.normal, scaled.normal]}>
        {this.props.children}
      </Text>
      );
  }
});

var scaled = StyleSheet.create({
  normal: {
    fontSize: width / scalingFactors.normal
  }
});

export default NormalText;
```

搞定啦！ `<HeadingText>` 组件使用了同样的方式，因此，无论什么时候，我们在应用的任何位置使用 `<HeadingText>` 和 `<NormalText>` 组件，它们的字体都会适当进行缩放。

9.6 小结及任务

Zebreto 应用可以作为一种参考。从多方面来说，它只是一个“最小可用的项目”，仍然还有很多可以提升的地方。话虽如此，代码中还是有不少值得探索之处，建议你可以深入挖掘它们。

如果想使用 React Native 进行更多实践，你可以看看 GitHub 仓库，然后扩展 Zebreto 应用。你可以从这些想法开始：

- 添加删除分组的功能；
- 添加一个可以查看分组中所有卡片的界面；
- 展示分组中卡片熟练值的信息；
- 尝试不同样式；
- 使用 ListView 改变分组组件。

下一章将介绍如何实际部署 Zebreto 或你自己的应用到应用商店中。

第 10 章

部署至iOS应用商店

我们已经完成了一个非常棒的应用，那么现在一定迫不及待地想把它交到用户手中了吧？这个步骤会根据平台而有所不同。本章将集中介绍部署应用到 iOS 应用商店的详细步骤。

作为 Web 开发者，我们习惯对部署过程有更强的控制。你可能习惯了一天上传多次代码到生产环境，并且版本通常都不是问题。但部署到 iOS 应用商店显然更加复杂，并且发布新版本通常需要 1~2 周的审核时间。因此，在计划阶段考虑好应用商店的提交和审核过程变得尤为重要。

10.1　准备Xcode工程

Xcode 项目中包含了许多关于应用的元数据。React Native 为你默认设置了一些参数，但在提交审核之前，我们需要确保某些属性已经设置正确。就 Zebreto 而言，我们的工程文件位于 iOS/Zebreto.xcodeproj。

如果由于某些原因，你还没有这么做的话，请尽快把 Xcode 工程文件加入版本控制。编辑工程时遇到 Xcode 崩溃，导致工程文件处于糟糕的状况，这类事件也不是没有耳闻。

在 Xcode 中打开工程（图 10-1）。你需要打开左面板，并关闭右面板和下面板（在右上角进行控制）。

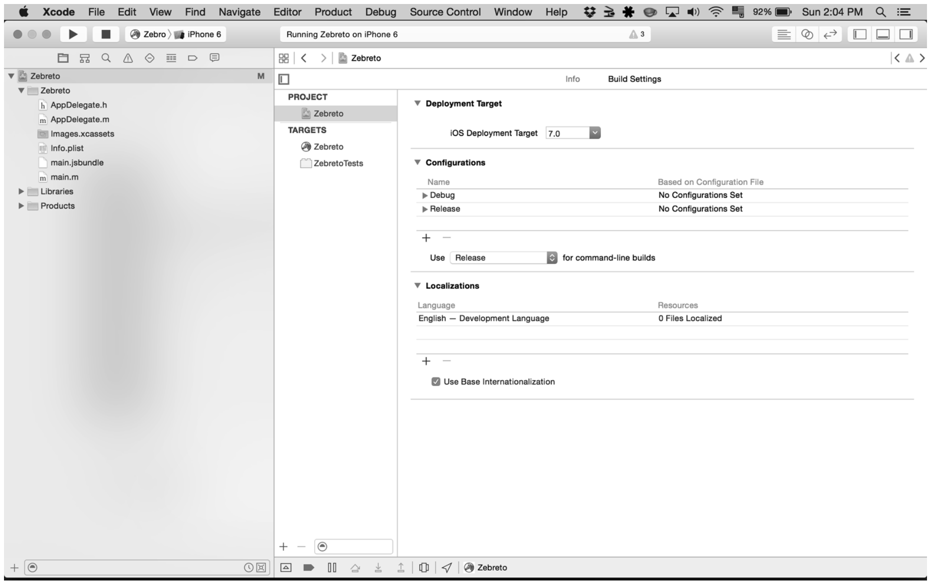


图 10-1：在 Xcode 中打开你的工程

10.1.1　选择支持的设备和目标iOS版本

你需要决定你的工程是以哪个 iOS 版本为目标的。让人困惑的是，这里有两个独立但是相关的设置：Base SDK Version（基础 SDK 版本）和 iOS deployment target（iOS 部署目标）。默认情况下，React Native 设置部署目标为 7.0，并使用最新的 iOS SDK（9.0）。部署目标指的是运行你的应用需要的最小的 iOS 版本，而 SDK 版本决定了你的应用会基于哪个 SDK 版本进行构建。这些设置的区别参见 Apple 文档（http://apple.co/1MVKVc3）。

我们目前只需要记住 Base SDK Version 需要大于或等于 iOS deployment target 即可。

如果你正在使用某些需要比默认设置更高版本的 iOS 接口，你需要正确地修改部署平台的设置，可以在工程的 Info 菜单中修改（图 10-2）。



图 10-2：选择目标 iOS 版本

如果你想更新 Base SDK 的版本，可以在 Build Settings 菜单中指定（图 10-3）。

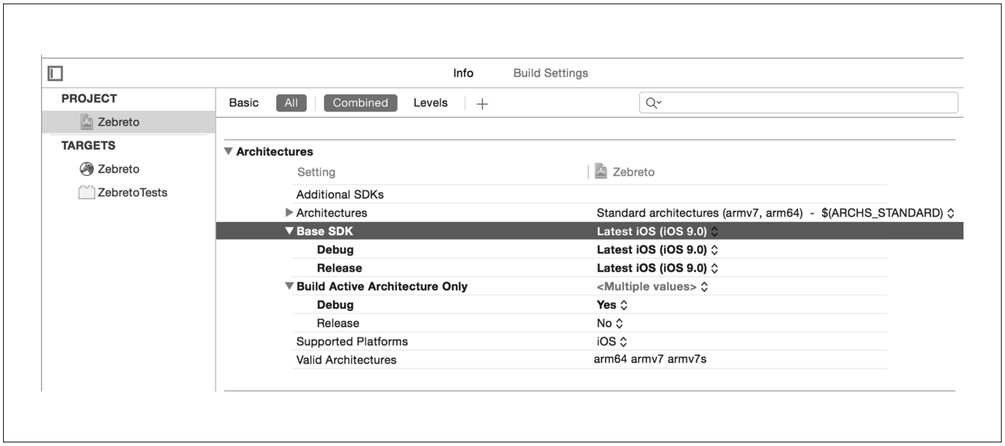


图 10-3：修改 Base SDK 版本

如果在 TARGETS 下选择了你的应用而不是 PROJECT 的话，你也可以指定应用支持的设备和屏幕方向。也就是说，对于 iOS 项目而言，你可以指定应用为适用于 iPhone、适用于 iPad 或者通用，通用意味着可以同时支持 iPad 和 iPhone 设备（图 10-4）。

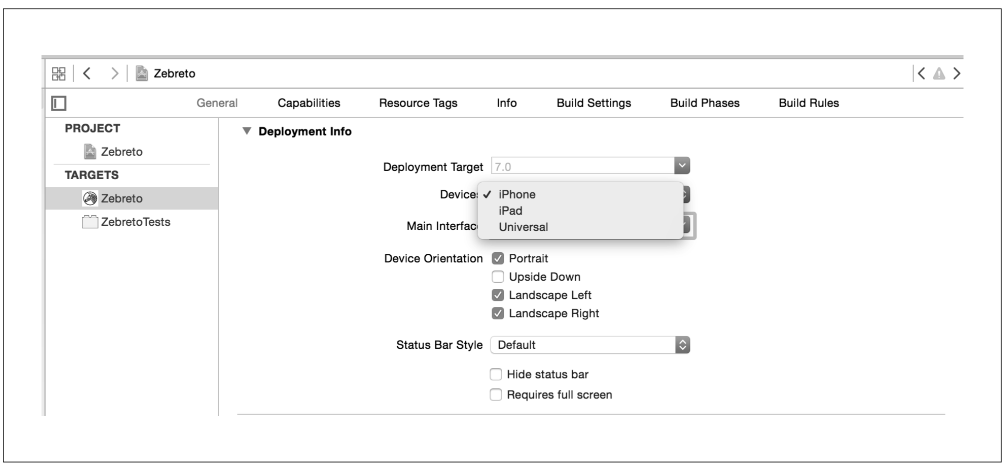


图 10-4：设置设备目标

选择支持的设备之后，我们就可以继续设置启动图像和应用图标了。

10.1.2　启动界面图像

启动界面图像是用户启动你的应用之后的加载过程中出现的占位图片。这里有多种适合的方法。有些采用“闪屏”的做法，并附带该应用的 logo 和名称。还有其他一些模仿应用的

用户界面，但是不带任何数据，实现了无缝过渡的效果。

不论你采用哪种方式，都需要提供所有支持设备的相应尺寸的启动界面图像。

首先，我们选择工程的 Image.xcassets/ 目录，然后创建一个新的图像集（图 10-5）。

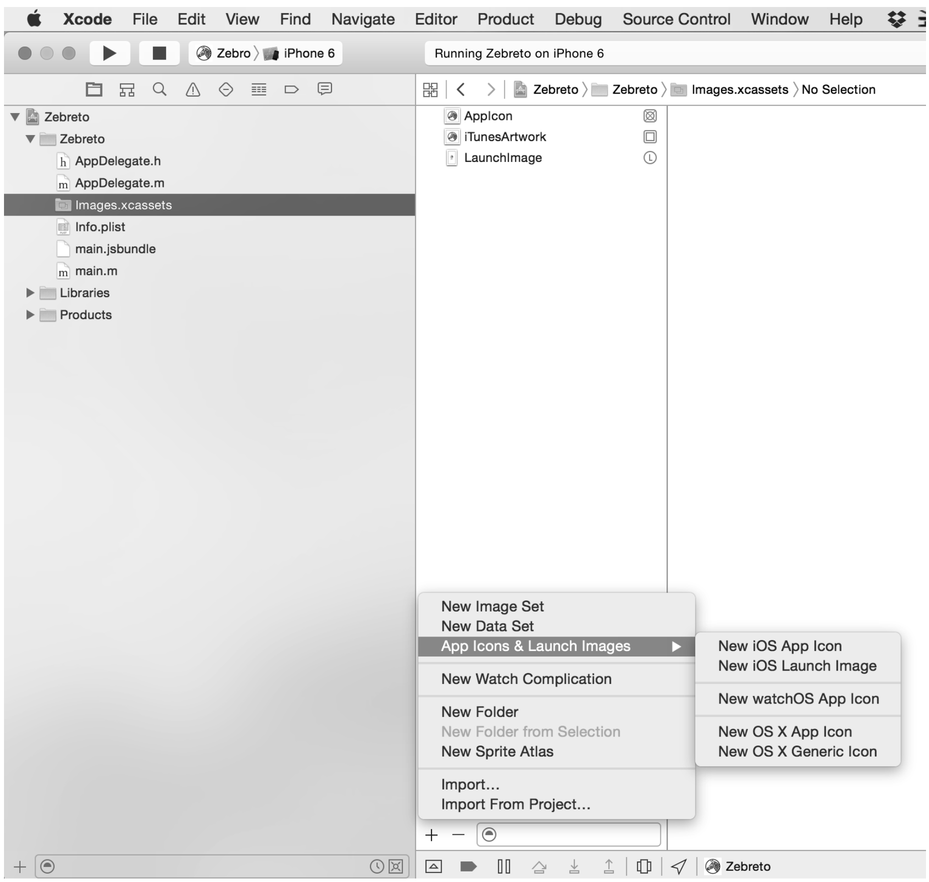


图 10-5：添加启动图像

在这里，你可以添加适应每个设备尺寸的启动图像（图 10-6），因此我们需要提供不少图片。

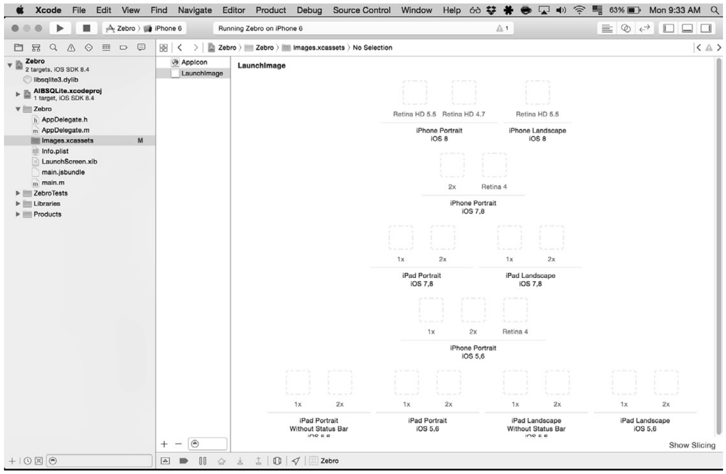


图 10-6：为每个设备尺寸和屏幕方向提供启动图像

所需的图片尺寸因设备而不同。例如，iPhone 6 的纵向模式需要 750 × 1334 像素的文件，而横向模式需要 1334 × 750 像素的文件。

特定平台所需的尺寸参见 Apple 文档（http://apple.co/1HaVNmb）的说明。如果你提供了错误尺寸的文件，Xcode 会给出警告，所以也可以利用警告来引导你。

10.1.3　添加应用图标

应用图标会展现在用户设备的主界面下，也会显示在应用商店里。就像启动界面一样，你需要提供适应支持设备正确尺寸的图标文件，可以在 Apple 文档查看尺寸（http://apple.co/1HaVNmb）。

Apple 的人机界面指南给出了一些基本的指导。应用图标应该不包含任何透明区域，并且必须为方形（Apple 自动帮你在图标上添加圆角效果，无需自己制作）。

就像之前设置启动图像一样，选择 Image.xcassets/ 目录之后，点击加号按钮。但是这次，我们选择创建一个应用图标（图 10-7）。



图 10-7：添加图标到工程中

你可将文件拖放到右边，把它们添加为图标。

如果你从模拟器或设备删除了应用，然后重新安装，那么应该可以看到应用图标，如图 10-8 所示。

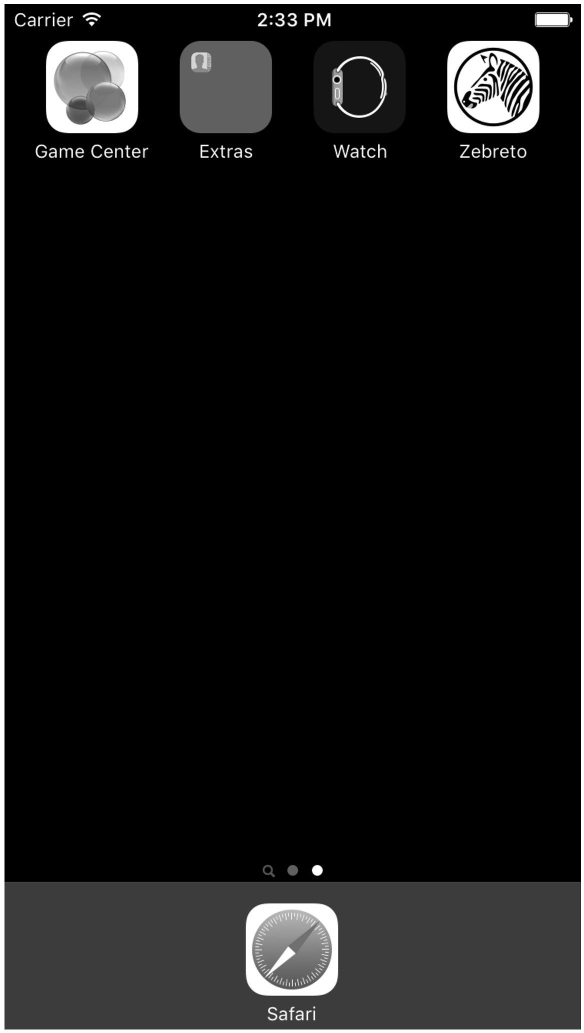


图 10-8：设置定制图标并重新安装之后，你可以在主界面看到图标，开发阶段也能生效

如果由于某些原因，你的启动图像或应用图标没有正确渲染的话，务必检查 App Icons 和 Launch Images 菜单下的设置，你可以在 General 设置菜单下找到它们（图 10-9）。

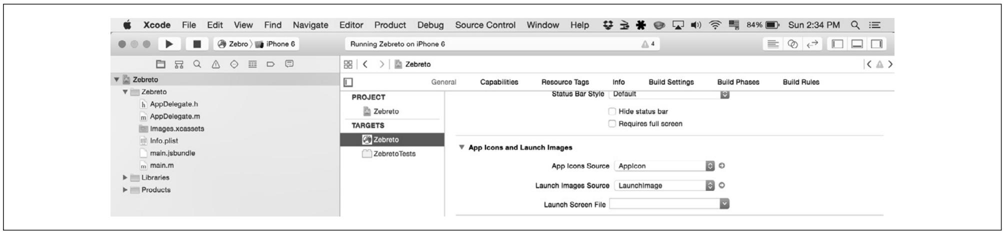


图 10-9：双击 App Icons Source 和 Launch Images 源文件

10.1.4　设置Bundle名称

工程的 Bundle 名称决定了你的应用在用户设备上将会显示的名称，所以非常重要。

注意，在重命名的时候 Xcode 可能会卡住，这有可能会损坏你的工程文件。所以在使用 Xcode 的重命名功能之前，请确保你的工程文件已经加入到版本控制系统中。

我们可以在右面板的 Identity and Type 里查看并修改名称（图 10-10）。这个名称是在你运行 `react-native init` 的时候自动设置的，如果你想修改它，就趁现在吧。

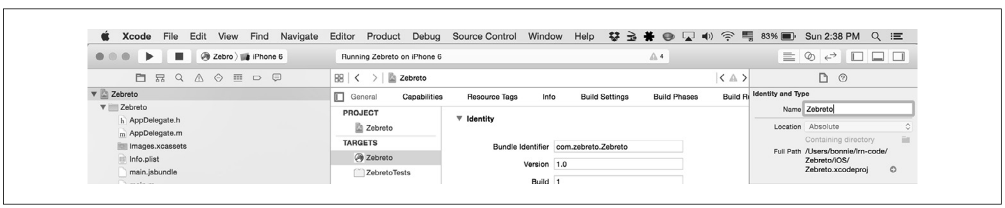


图 10-10：Identity and Type 菜单下的名称是面向用户的

10.1.5　更新AppDelegate.m

回想一下 AppDelegate.m 文件的作用，该文件中有两种指定 JavaScript 代码位置的方法：从打包的文件加载，或通过 localhost 加载。在开发期间运行 React Native 包管理器是推荐的做法，但对部署来说，需要生成一个打包之后的 JavaScript 文件。

第一个选项使用 localhost，我们将其注释，然后启用第二个选项，从打包文件加载代码：

```
// jsCodeLocation =
// [NSURL URLWithString:@"http://localhost:8081/index.ios.bundle"];

...

jsCodeLocation = [[NSBundle mainBundle]
URLForResource:@"main" withExtension:@"jsbundle"];
```

搞定之后，我们需要生成打包文件。

在项目目录运行：

```
react-native bundle --minify
```

我们需要确保这样修改之后还可以在模拟器运行。重启模拟器，然后尝试运行应用，它应该能如常运行。

10.1.6　为发布设置Schema

接着，我们需要将 Build Scheme 设置为 Release，而不再是 Debug。如图 10-11 所示，找到 Product → Scheme → Edit Scheme... 菜单选项。

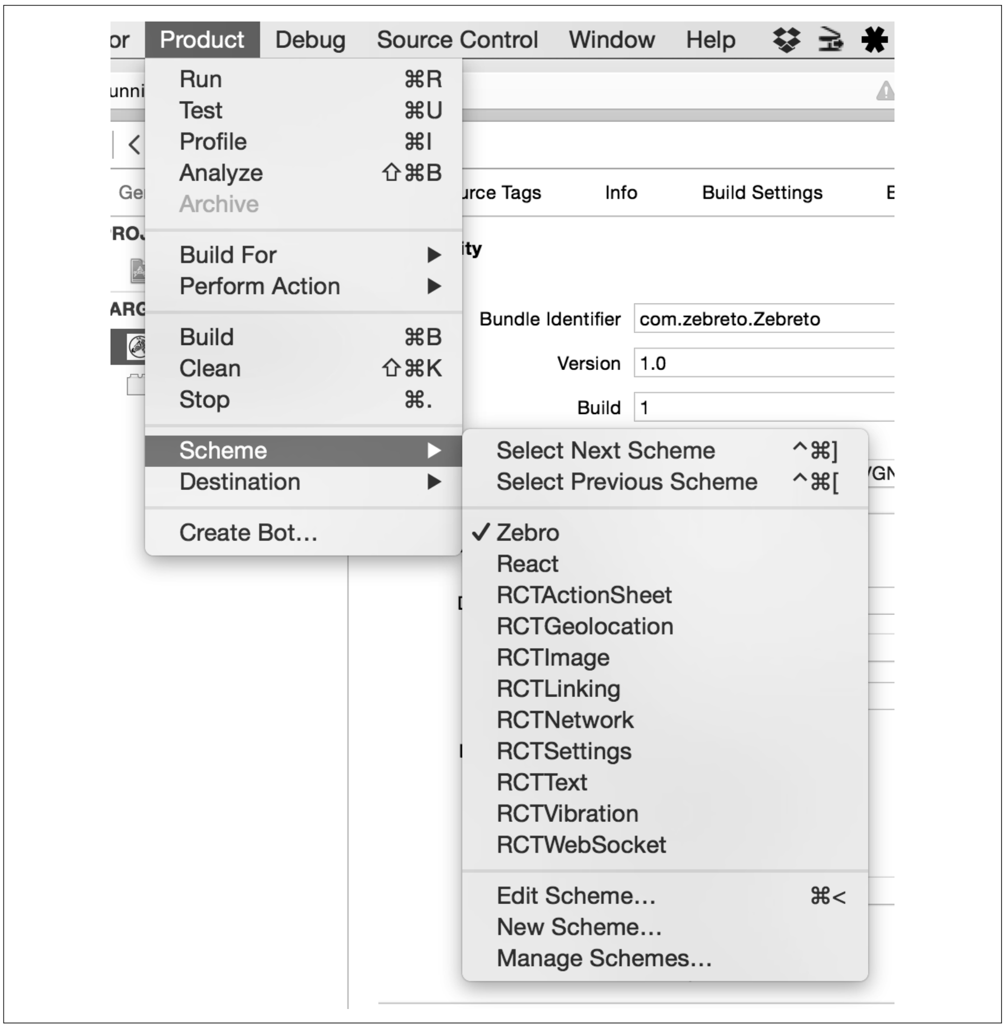


图 10-11：选择 Edit Scheme... 选项

然后修改项目的 Schema 为 Release（图 10-12）。

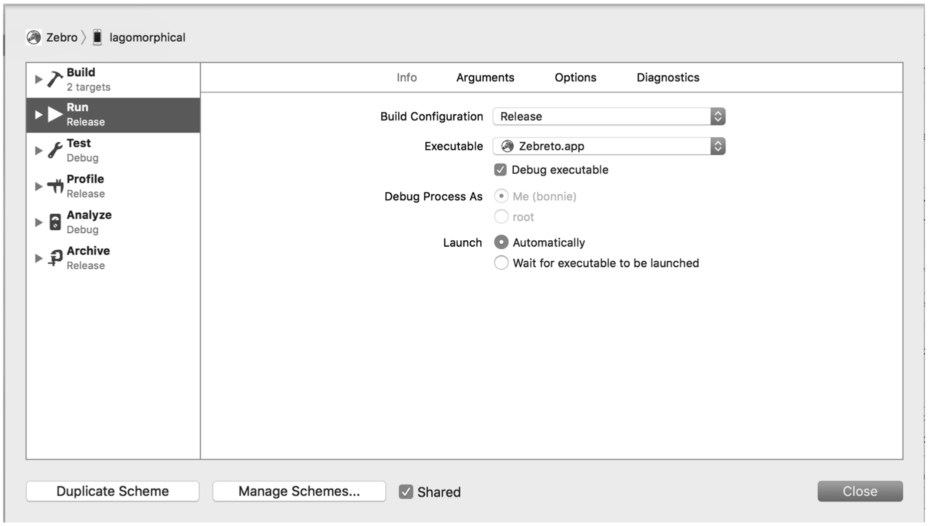


图 10-12：设置 Build Configuration 为 Release 并取消勾选 Debug executable

这样修改之后，意味着运行应用时调试菜单就不会再出现了。

10.2　上传应用

好了，既然你的项目已经正确配置成发布状态了，现在是时候把它提交到 Apple 应用商店了！

10.2.1　完成协议文书

如果没有 Apple 开发者账号是不能提交到应用商店的，所以如果你还没有注册，现在可以行动了！申请账号的价格是每年 99 美元。

此外，如果你以前申请过的话，现在也需要签署更多的新协议。与此相关的错误提示信息有时候很难理解，如图 10-13 所示。

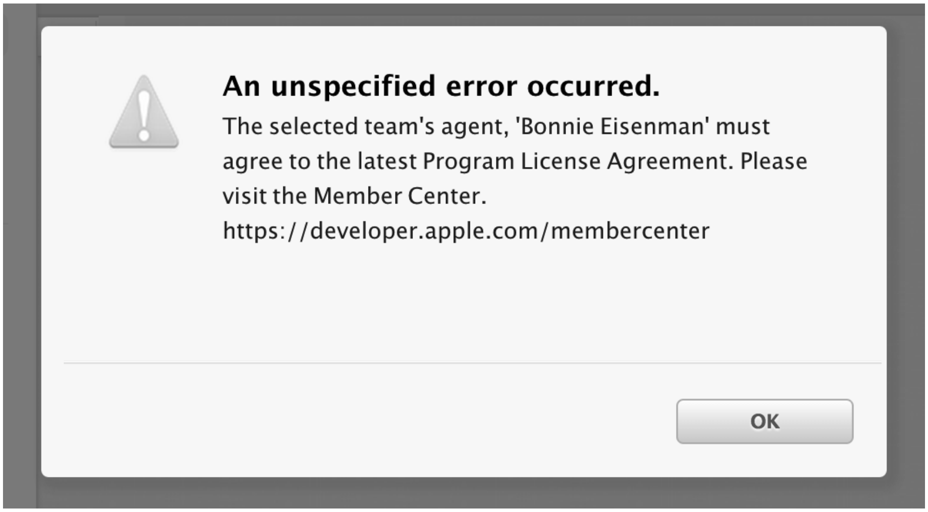


图 10-13：如果看到此类错误，先尝试遵循提供的指示

不幸的是，有时候访问特定的网址（https://developer.apple.com/membercenter）也会出现类似的错误，如图 10-14 所示。

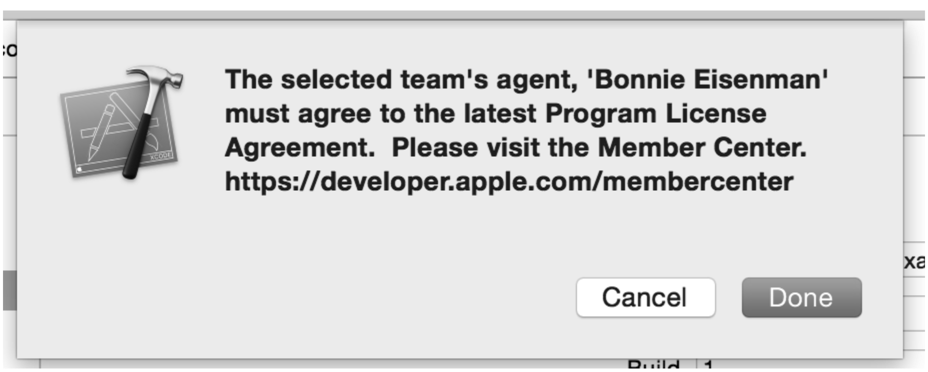


图 10-14：如果你看到此类错误，访问 iTunes Connect 并检查是否有未完成的协议

如果碰到这类问题，先访问 iTunes Connect（https://itunesconnect.apple.com/），而不是会员中心。选择 Agreements,Tax,and Banking，然后填写所有未完成的表格。

10.2.2　创建归档

下一步需要做的是创建应用的归档，创建归档后才能提交至应用商店。你可以在 Product → Archive 菜单里找到它。

如果 Archive 菜单显示为灰色禁用状态（如图 10-15 所示），那么很可能是因为你选择了 iOS 模拟器作为构建目标。

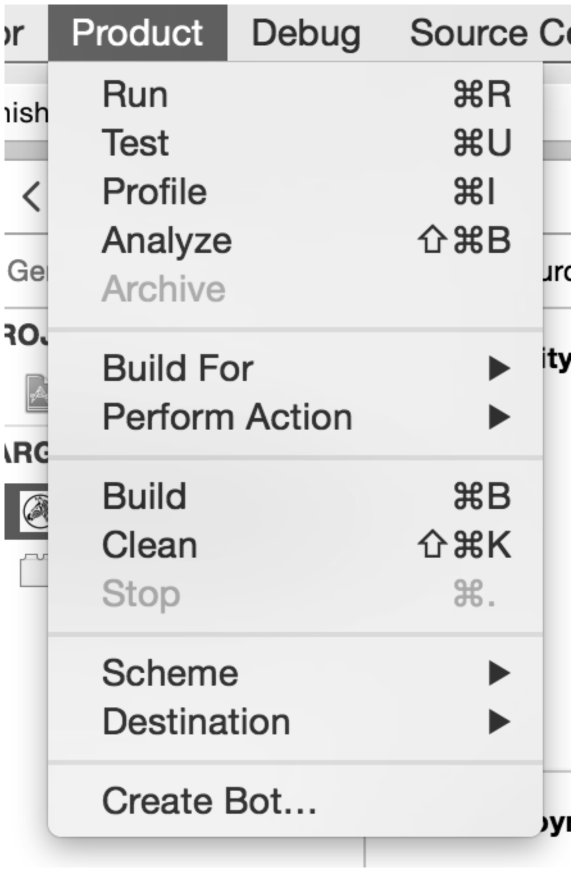


图 10-15：如果 Archive 选项被禁用，尝试修改构建目标

将工程的构建目标修改为 iOS 设备之后，就可以在菜单上选择 Archive 选项了（图 10-16）。

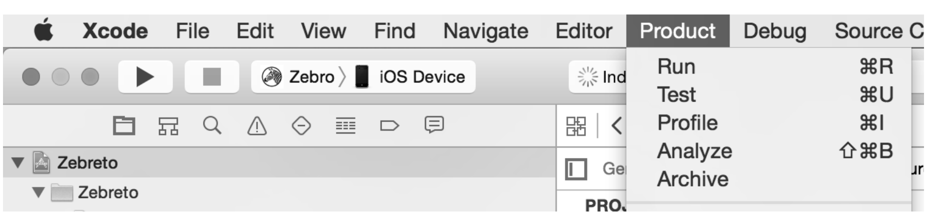


图 10-16：选择 Product → Archive，开始创建归档

如果顺利的话，将会出现 Archive 的界面（图 10-17）。

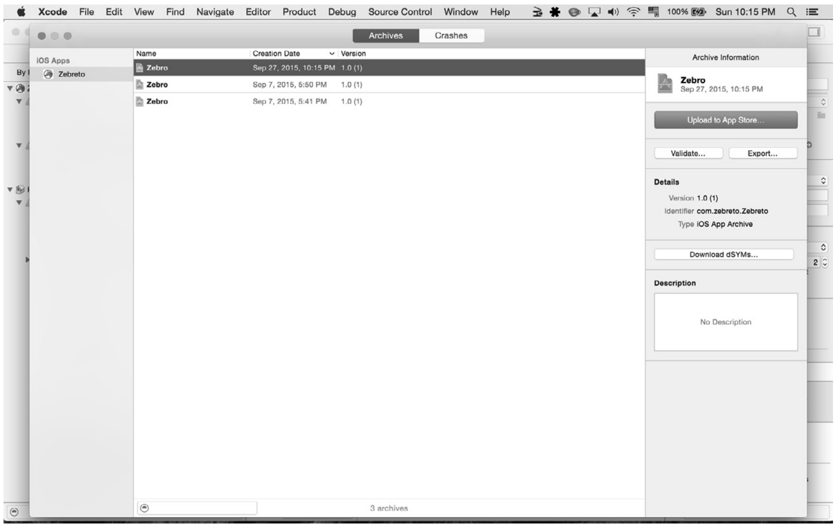


图 10-17：选择一个归档，上传至应用商店

作好准备点击 Upload to the App Store 的按钮了吗？快行动吧！ Xcode 将最后进行一些检查，然后你的应用就被提交到 Apple 应用商店了（图 10-18）。

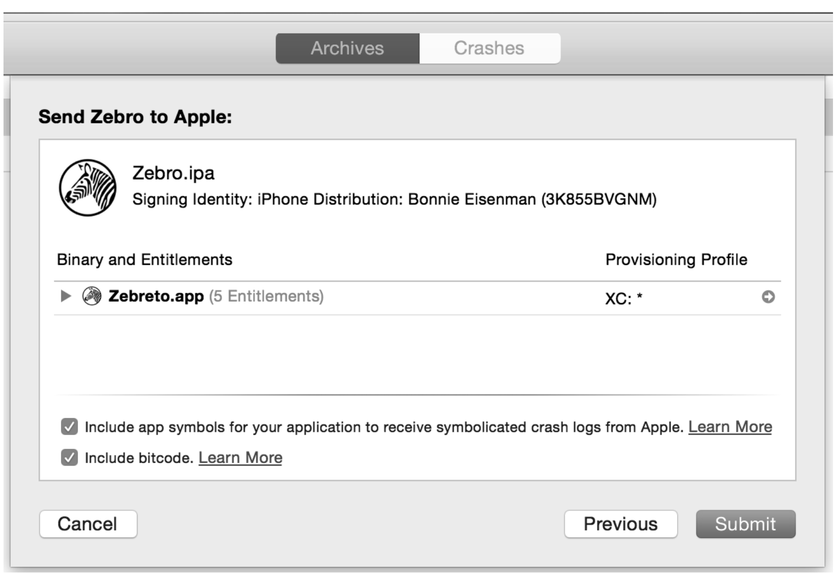


图 10-18：点击 Submit 按钮，上传归档至应用商店

10.2.3　在iTunes Connect上创建应用

如果你觉得已经大功告成了，那么很抱歉告诉你，依然还有工作要做！你的应用归档已经被上传了，但你现在需要通过 iTunes Connect 开始真正的提交。这个过程包括提交一些重要的关于应用的元数据信息，例如应用描述和屏幕截图，这些都是呈现给用户的。

想要获取该流程的更多深入的信息，可以在 Apple 文档（http://apple.co/1S6zsXq）中查看关于如何创建 iTunes Connect 记录的相关内容。

首先，你需要在开发者中心（http://apple.co/1NLquel）注册一个 App ID。这个表格需要你输入应用的 Bundle Identifier（图 10-19）。



图 10-19：在 Xcode 中设置 Bundle Identifier

此处的信息需要跟你的 Xcode 项目中的 Bundle Identifier 保持一致（在 Identity 菜单下），如图 10-20 所示。

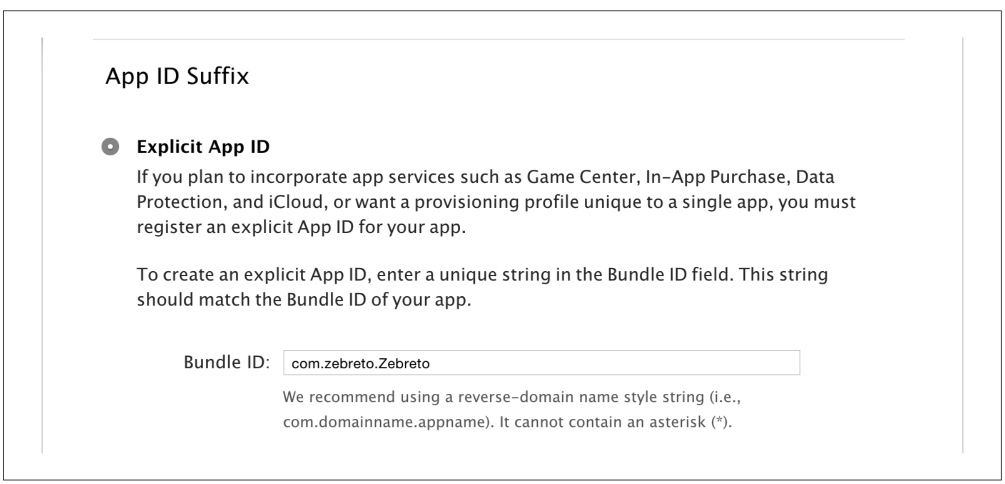


图 10-20：开发者中心的 Bundle ID 需要匹配 Xcode 中的 Bundle Identifier

> **Bundle Identifier**
>
> 如果你的 Bundle Identifier 不匹配，那么就不能把应用归档和 iTunes Connect 中的记录联系起来。请仔细检查一下！

接下来，我们在 iTunes Connect（https://itunesconnect.apple.com/）上创建一个新的应用（图 10-21）。

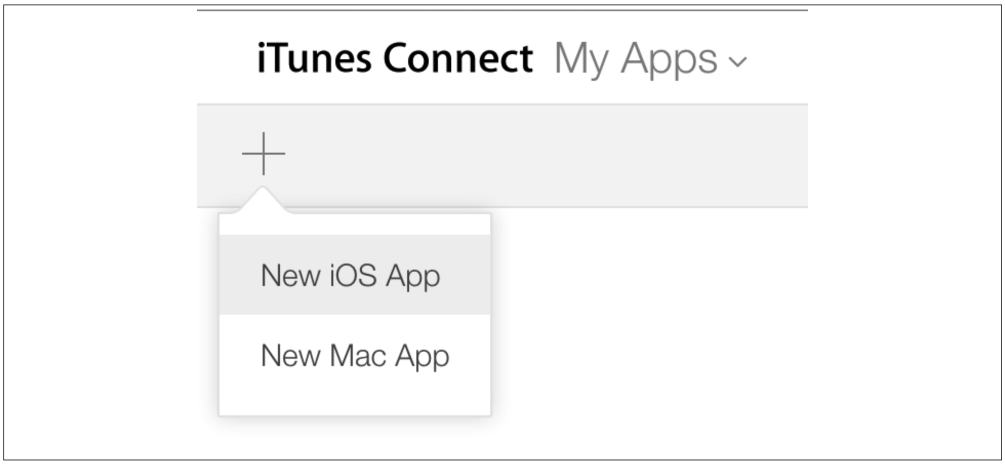


图 10-21：在 iTunes Connect 上选择 New iOS App 来创建一个新的应用

Bundle Identifier 再一次出现了，选择合适的那个，然后继续创建应用。

如果应用归档已经成功上传，那么可以在 iTunes Connect 看到构建的项目（图 10-22）。

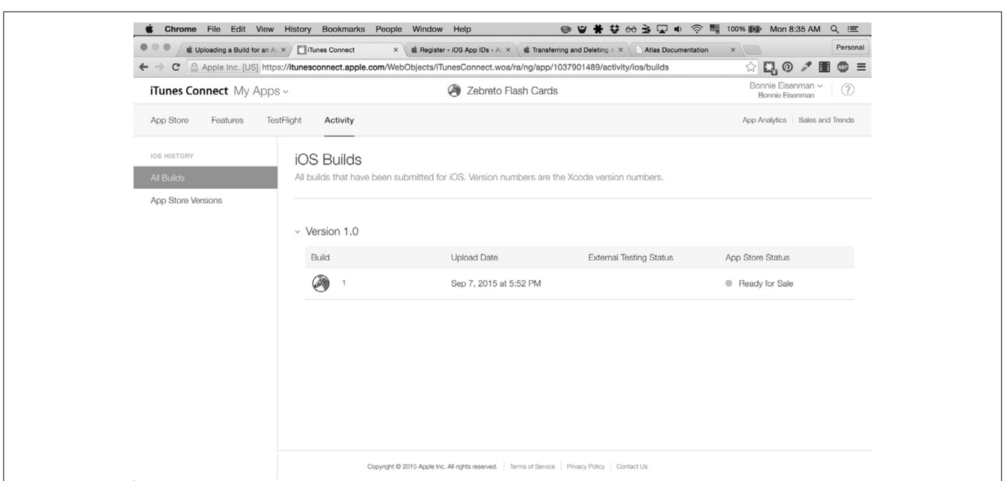


图 10-22：应用归档上传之后，会出现在构建项目列表中

如果你没有看到任何构建项目，请检查下列事项。

- iTunes Connect 中的 Bundle ID 是否与 Xcode 项目保持一致？
- 在上传归档时，Xcode 是否有出现任何错误？
- 如果重新上传应用归档，会有什么反应？

现在，在 iTunes Connect 中你可以输入应用商店列表相关的信息（例如正确的分类和描述，等等）。这里有很多信息需要填写，别着急。

重要的是：这个页面也可以让你上传屏幕截图和应用漫游的视频。提供高质量的屏幕截图是使你的应用在应用商店里脱颖而出的关键举措。通常你需要为不同设备提供相应尺寸的屏幕截图（图 10-23）。

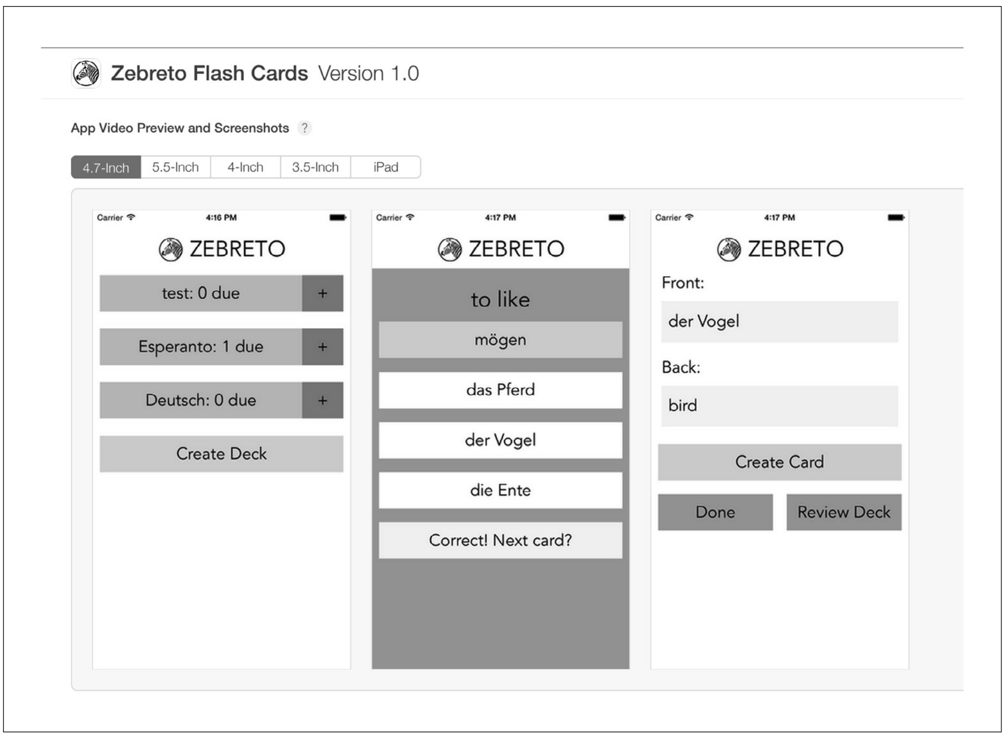


图 10-23：上传屏幕截图

iOS 模拟器上的屏幕截图

使用 iOS 模拟器可以轻松获得适合尺寸的屏幕截图。加载每一种设备，然后按下快捷键 Command+S，就可以保存一张屏幕截图了。

10.3 使用TestFlight进行Beta测试

在提交应用商店审核之前，你应该使用 TestFlight 进行 Beta 测试。即使你是唯一的一个“Beta 测试者”，使用 TestFlight 而不是开发模式下的应用，可以让你更精准地把握你的应用初次安装到用户设备时的使用体验。

TestFlight 可以让你轻松地发送测试邀请给用户。在 iTunes Connect 上的申请记录下面，选择 TestFlight，然后添加 Beta 测试者（图 10-24）。你需要提供他们的电子邮件地址。

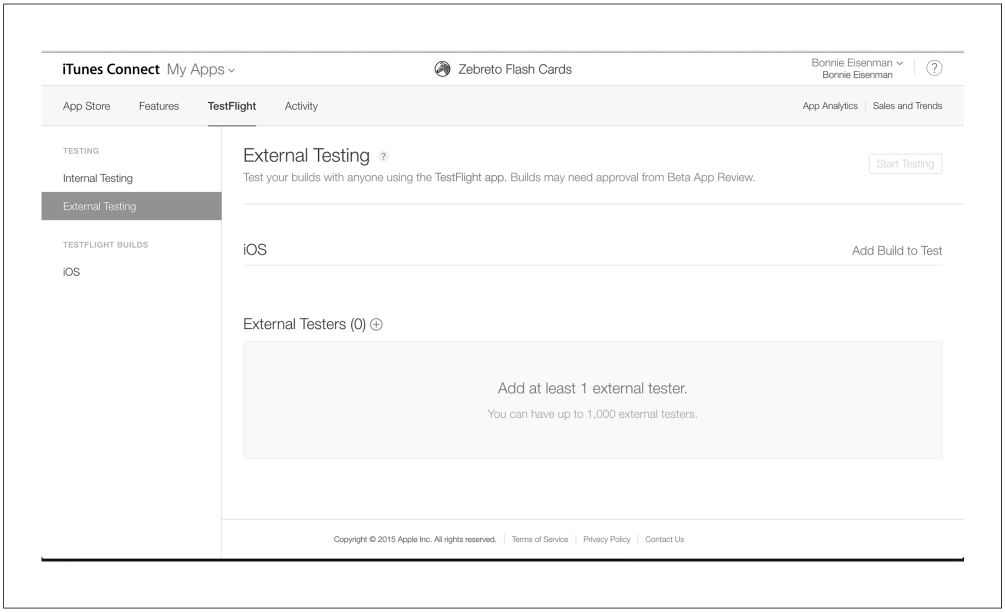


图 10-24：iTunes Connect 中的 TestFlight 界面

你的 Beta 测试者初次使用时需要安装 TestFlight 应用。当他们接收到测试应用的邀请后，TestFlight 会给他们显示安装应用的选项（图 10-25）。成功安装之后，应用就可以正常运行了。

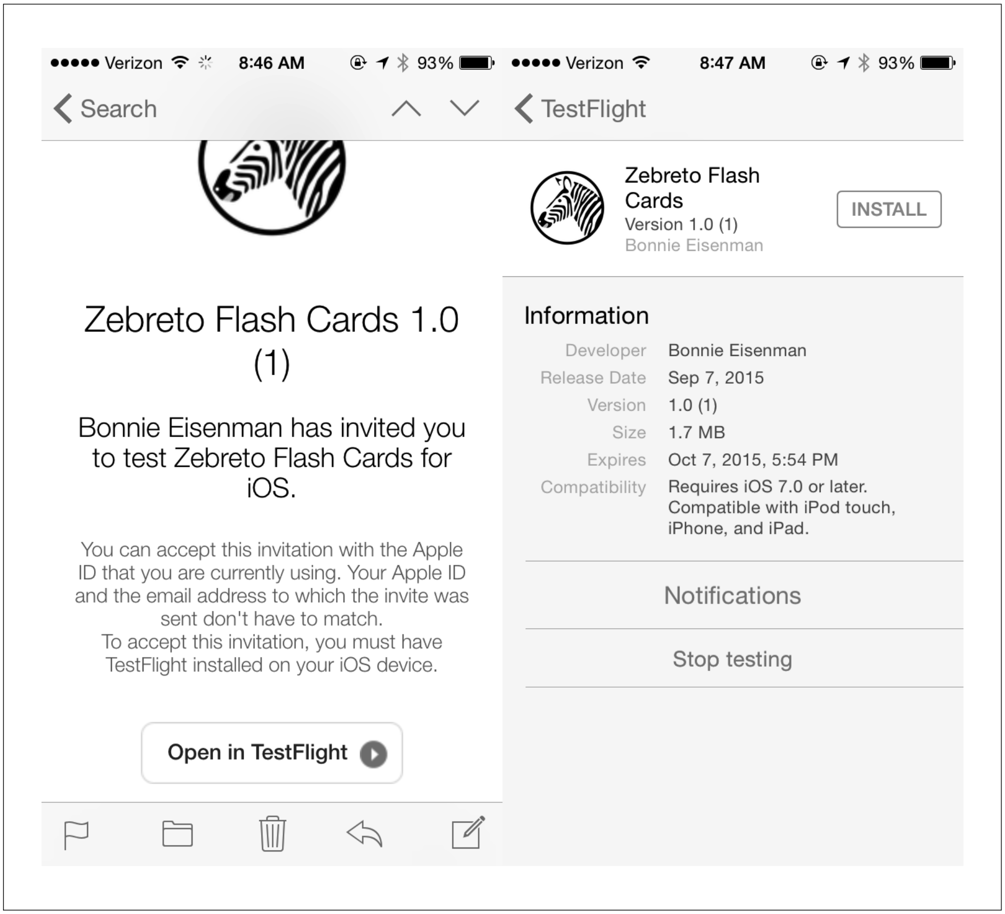


图 10-25：Beta 测试者通过 TestFlight 接收邀请邮件，然后安装应用

10.4　提交应用审核

当你得到了 Beta 测试者满意的反馈，并且已经填写了 iTunes Connect 上的相关信息之后，现在你（终于！）可以提交应用审核了。提交之后，iTunes Connect 将会标识你的应用状态为“等待审核”（图 10-26）。



图 10-26：应用状态可在 iTunes Connect 上查看

应用进入审核流程后，审核的每一环节你都会收到邮件提醒，无论是被拒绝还是被接纳。平均来说，应用商店的审核流程大概是 1~2 周，但不好说你的应用审核需要多少时间。如果恰逢一年之中繁忙的时段，如节假日，审核周期可能会相应延长。

这里提供一个参考：Zebreto 应用等待 8 天之后就被接纳了。当你的应用被接纳之后，祝贺你，现在可以在应用商店下载它了。

10.5 小结

我们辛辛苦苦完成了应用，将它交付给用户的那一刻一定很振奋人心吧？然而发布应用只是一个开始，因为你需要不断支持应用的后续发布。部署应用不像在 Web 平台上那样频繁且易于操作；发布 iOS 的新版本需要花费时间，并且它有更长的寿命。许多 iOS 用户没有开启自动升级，所以每一个版本都很重要。至少你每次提交更新或修复 bug 都需要等待 Apple 的审核。（对于非常严重的 bug，你可以申请加急，但要谨慎使用这个功能！）

此外，iOS 应用的发布从评分的角度来看有些风险。展示在应用商店页面上的评论是基于当前版本的，而不是整体的评分，所以如果发布了一个有 bug 的版本，那么将很可能为你的应用带来很大的负面影响。记住，测试很关键！

如果你有新版本要提交的话，流程跟最初上传的流程非常相似。在 Xcode 中增加你的应用版本号，然后提交一个新的归档。你可以在 iTunes Connect 上找到提交审核新构建版本的选项。

本章已经介绍了如何提交应用至 iOS 应用商店，下一章我们将把视线转移到 Android 平台，并操作相似的流程。

第 11 章

部署Android应用

终于到这一步了！准备好部署 Android 应用然后交到用户手中了吗？如果你已经体验过 iOS 应用的提交流程，那么这里的步骤会很相似。不过值得庆幸的是，Play 商店的审核流程会更简单，审核速度也会更快：你只需要等待 1~2 个工作日就可以知道审核结果。

本章将介绍如何生成 React Native 应用的部署版的 APK，以及如何将其分发给 Beta 测试者并提交至 Goole Play 商店审核。

> **查看文档**
>
> 这里我们会作一个关于部署 Android 应用的详细指导，但请把官方文档（https://facebook.github.io/react-native/docs/signed-apk-android.html）作为最新的流程参考。

11.1　设置应用图标

虽然 Android 自带的图标挺可爱的，但你在部署之前一定想换成自己定制的应用图标。

在 android/app/src/main/AndroidManifest.xml 文件里指定应用图标：

```
android:icon="@mipmap/ic_launcher"
```

文件路径指向了 android/app/src/main/res/ 目录。第 3 章已经介绍过，Android 图像资源根据分辨率放置在不同目录下。图标文件也一样，你会发现这里已经放置了默认的图标（图 11-1）。

```
res
├── mipmap-hdpi
│   └── ic_launcher.png
├── mipmap-mdpi
│   └── ic_launcher.png
├── mipmap-xhdpi
│   └── ic_launcher.png
├── mipmap-xxhdpi
│   └── ic_launcher.png
├── mipmap-xxxhdpi
│   └── ic_launcher.png
```

图 11-1：应用图标文件的目录结构

可以直接替换这些文件，或在 AndroidManifest.xml 文件中修改图标路径。在设备上重新安装应用之后，就可以看到新的图标了（图 11-2）。

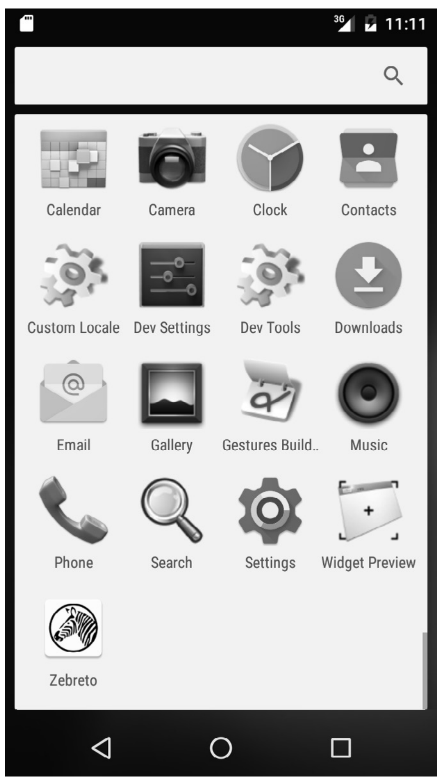


图 11-2：查看已安装的应用，现在可以看到 Zebreto 定制的图标

创建不同分辨率的图标听起来可能有些枯燥，我们不妨使用工具来生成适合的尺寸。我个人是 romannurik 项目的拥护者（https://romannurik.github.io/AndroidAssetStudio/icons-launcher.html）。

11.2 生成release版本的APK

为了部署 Android 应用，我们需要生成 release 版本的 APK（APK 文件是 Android 应用安装包文件，该格式用于发布 Android 应用）。通常来讲，有 5 个基本步骤：

(1) 生成签名密钥；
(2) 设置 gradle 变量；
(3) 添加签名到应用的 gradle 配置中；
(4) 生成 release 版本的 APK；
(5) 在设备上安装 release 版本的 APK。

我们将会逐一讲解这些步骤。你可以阅读 Android 官方的发布概要（http://developer.android.com/tools/publishing/publishing_overview.html）来获得更多信息。

首先，需要为你的应用生成一个签名密钥。

你可以使用 `keytool` 来生成一个 keystore（密钥对）和 key（密钥）：

```
$ keytool -genkey -v -keystore my-release-key.keystore \
    -alias my-key-alias -keyalg RSA -keysize 2048 -validity 10000
```

Android 文档（http://developer.android.com/tools/publishing/app-signing.html）中有更多关于应用签名的说明。Android 使用证书签名来识别应用的作者。不要忘记密码，也不要丢失密钥。同时具备这两个信息才能对应用进行升级。

之前的命令将会生成一个 my-release-key.keystore 文件。将它移动到项目的 android/app/ 目录下。

接着，新建或编辑 ~/.gradle/gradle.properties 文件并添加如下代码，如例 11-1 所示。

例 11-1：添加这些变量到 ~/.gradle/gradle.properties 文件中

```
MYAPP_RELEASE_STORE_FILE=my-release-key.keystore
MYAPP_RELEASE_KEY_ALIAS=my-key-alias
MYAPP_RELEASE_STORE_PASSWORD=*****
MYAPP_RELEASE_KEY_PASSWORD=*****
```

然后把星号替换成之前调用 `keytool` 时使用的密码。

在 ~/.gradle/gradle.properties 文件中添加这些信息之后，我们就成功地把它们加入到 gradle 通用配置中了（记住，gradle 是用来构建 React Native 项目的工具）。

保管好密钥

千万不要把你的密钥密码加入到版本控制中，也不要丢失密钥！在发布应用之后，如果你使用了一个新的密钥，那么会刷新 Play 商店上的数据，所有的下载统计和评论也会因此而丢失。

现在我们已经设置好了 gradle 变量，接着需要把我们的签名配置添加到应用的 gradle 配置中。打开 android/app/build.gradle 文件，然后添加如下签名配置（例 11-2）。

例 11-2：修改 android/app/build.gradle

```
...
android {
  ...
  defaultConfig { ... }
  signingConfigs {
    release {
      storeFile file(MYAPP_RELEASE_STORE_FILE)
      storePassword MYAPP_RELEASE_STORE_PASSWORD
      keyAlias MYAPP_RELEASE_KEY_ALIAS
      keyPassword MYAPP_RELEASE_KEY_PASSWORD
    }
  }
  buildTypes {
    release {
      ...
      signingConfig signingConfigs.release
    }
  }
}
...
```

注意，这里使用了之前在 ~/.gradle/gradle.properties 文件中定义的变量。

好了！现在开始生成签名版的 APK。

进入项目根目录，在终端运行 React Native 包管理器：

```
$ npm start
```

然后在根目录再次运行下列命令：

```
$ mkdir -p android/app/src/main/assets
$ curl \
"localhost:8081/index.android.bundle?platform=android&dev=false&minify=true" \
    -o "android/app/src/main/assets/index.android.bundle"
$ cd android && ./gradlew assembleRelease
```

哇！这里发生了什么？首先，我们新建了一个 assets/ 目录来存储打包的 JavaScript 文件。然后，通过 `curl` 从 React Native 包管理器获取了打包的 JavaScript 文件。最后，使用

`gradlew` 来构建 release 版本的 APK。

这个流程可能会改变

React Native 团队已经指出，这个流程可能会在将来的 React Native 中有所改动，因为使用 `curl` 从某个特定 URL 获取文件的方式并不是最直观的。记住，永远以官方文档为准（https://facebook.github.io/react-native/docs/signed-apk-android.html）。

接下来，你可以关闭 React Native 包管理器。打包的 JavaScript 文件已经被保存到硬盘中了。

在项目的 android/ 目录下，运行下面的命令来安装签名版的 APK:

```
./gradlew installRelease
```

这个命令将会安装签名版的 APK 到你的设备中。

永远记得在部署之前测试你的应用。开始的时候，可以使用 `gradlew installRelease` 命令上传这个 APK 到模拟器或连接的物理设备中。

11.3 通过邮件或链接发布

你知道吗？实际上，将应用发布给 Android 用户不一定需要上传到 Play 商店。在必要的时候，或者为了测试，你可以直接通过电子邮件将 APK 发布给用户。在 Android 设备上打开电子邮件就可以安装了。

你的 APK 文件位于 android/app/build/outputs/apk/app-release.apk。你可以通过检查下列文件存在与否来判断 release 版本的 APK 是否已经成功构建：

```
$ ls android/app/build/outputs/apk/
app-debug-unaligned.apk  app-debug.apk
app-release-unaligned.apk  app-release.apk
```

通过邮件将这个文件发布给用户，可以让他们下载并安装应用。事实上，用户在任何地方通过 Android 平台打开 APK 链接都可以安装你的应用。

注意：首先需要允许从未知来源来装应用。查看“Android 关于未知来源的文档”获取更多信息（http://developer.android.com/tools/publishing/publishing_overview.html#unknown-sources）。

11.4 提交应用至Play商店

在真实设备上（如果可能的话，在多个设备上）测试 release 版本的 APK 一段时间之后，

你打算将它部署到 Google Play 商店。那就开始吧！

这个过程相当容易，并且审核速度也很快，通常可以在提交之后的 24 小时之内部署成功。

进入 http://developer.android.com 网站，然后在页面右上角点击 Developer Console 按钮（图 11-3）。

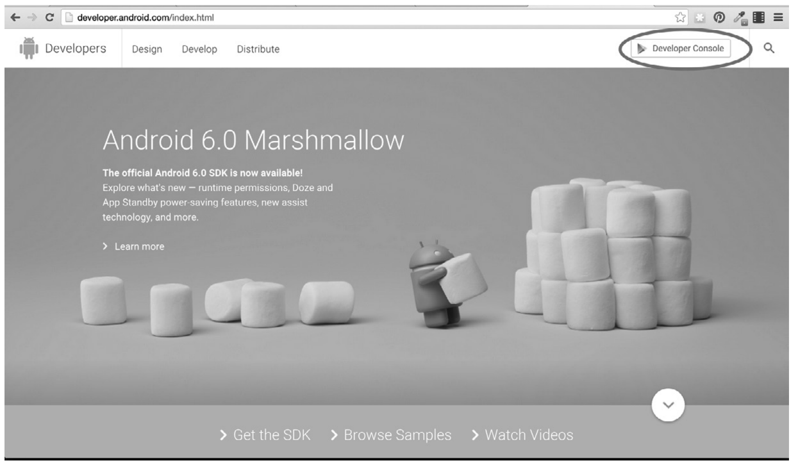


图 11-3：从 http://developer.android.com 进入 Developer Console

如果你还没有开发者账号，先申请一个，然后接受 Google 所有的条款。接着，点击左侧菜单上的 Android 图标来查看应用菜单。

点击 +Add new application 按钮，创建应用（图 11-4）。

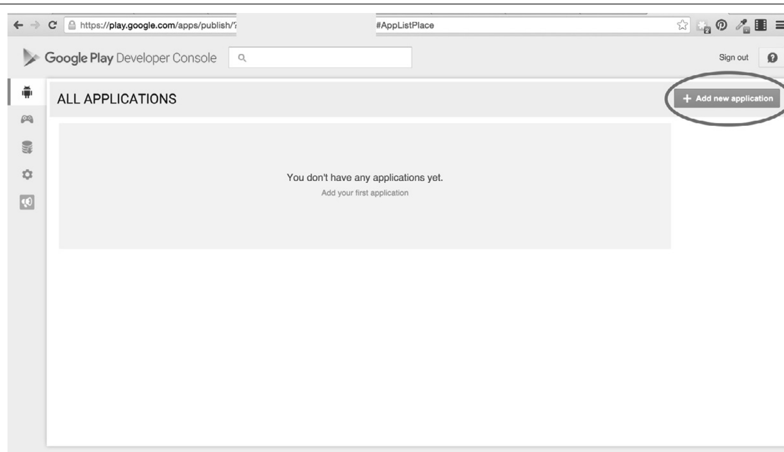


图 11-4：创建一个新的应用

在这里，你可以选择先上传一个 APK 文件，或编辑你的 Play Store 列表。我们选择其中一个选项。

上传你的 APK 文件，需要在文件系统里找到 release 版本的 APK 文件，它应该在这里：android/app/build/outputs/apk/app-release.apk（图 11-5）。

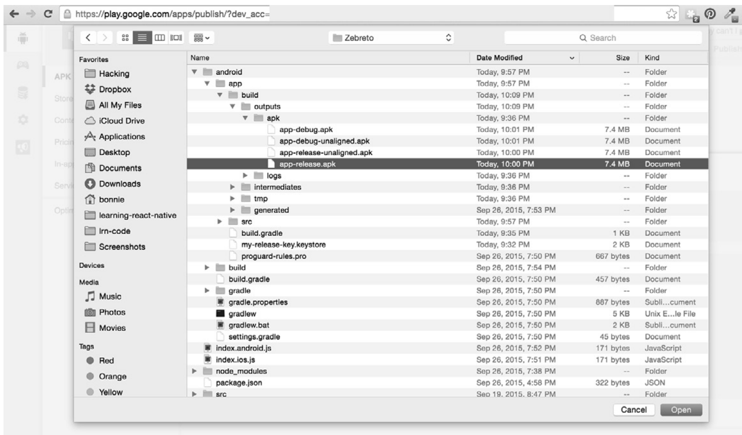


图 11-5：选择你的 app-release.apk 文件

APK 文件上传之后，你可以完成余下的 Play Store 表单，或建立一个 Beta 测试。

11.4.1　通过Play Store进行Beta测试

Google Play 商店提供了一个简单的 Beta 测试功能。上传 APK 文件之后，你可以选择 Beta Testing 选项卡，然后添加 Beta 测试者（图 11-6）。

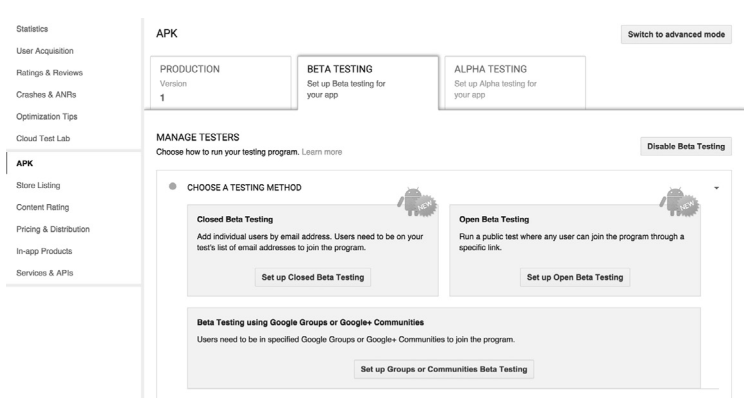


图 11-6：Play 商店的 Beta Testing 选项

这里你可以有几种选择。

- 开放Beta测试

  该选项允许用户通过特定链接加入测试。

- 封闭Beta测试

  该选项允许通过电子邮件地址添加单独的用户。

- 通过Google Groups或Google+社区进行Beta测试

  该选项允许特定的 Google Group 会员加入 Beta 测试。

Google 让你可以轻松地发布 APK 给这些用户。

对于 Android 平台而言，你需要把应用分发给尽可能多的用户。鉴于市面上有很多不同的 Android 设备，这一点甚至比 iOS 更为必要。例如 Zebreto 这一使用 React Native 默认设置的应用，在 Play 商店列表展示了所兼容的 7867 种不同的设备（图 11-7）。

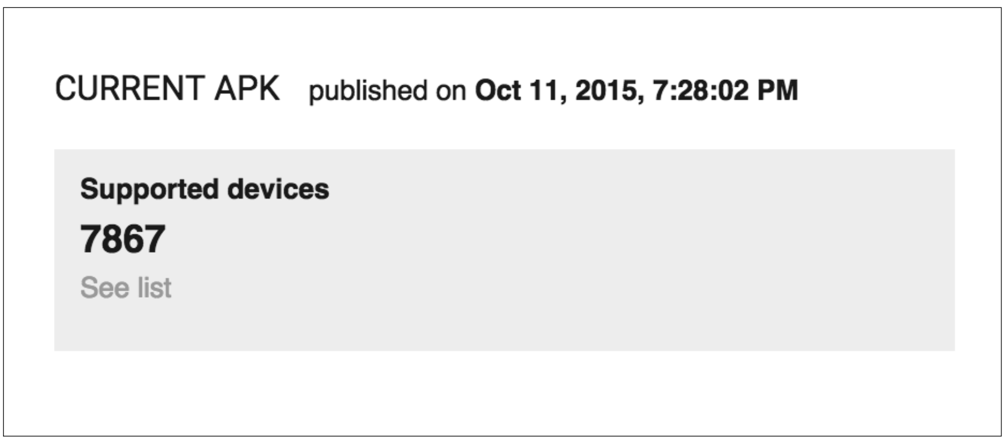


图 11-7：Play 商店会列出你的应用支持哪些设备

不同的设备有不同的尺寸、分辨率以及不同的特性，甚至不同的样式。一些制造商在 Android 默认的 UI 上定制了他们自己的皮肤。所以，好好利用 Beta 测试功能吧！

11.4.2 Play商店列表

你的 Play 商店列表包含了你的应用的重要信息（例如名称、描述、类别和内容分级，等等）。为了发布应用，你需要填写其中的大部分信息。

点击 Why can't I publish? 链接，Developer Console 将会帮助你列出需要完成的余下的任务（图 11-8）。

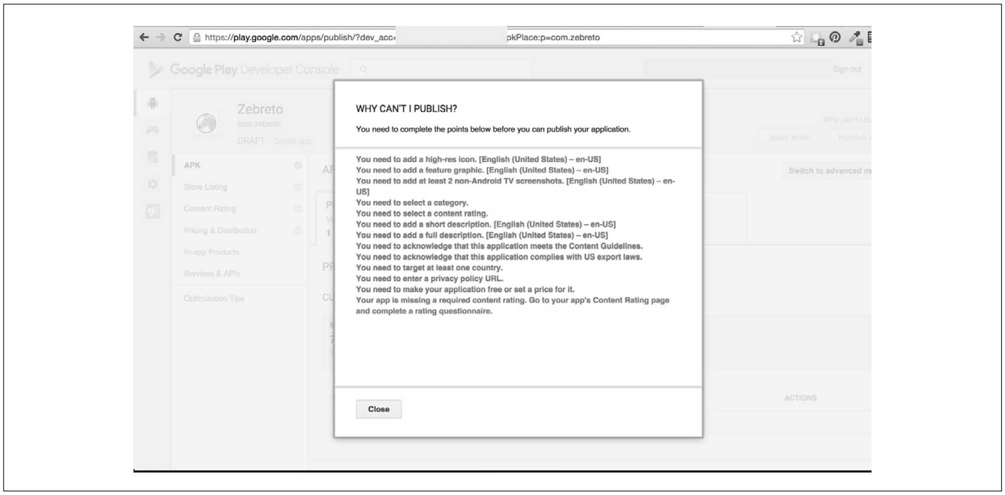


图 11-8：在发布前需要完成余下的任务

当你完成某项所需的任务之后，左侧菜单的对号将会变成绿色。完成所有必要的步骤之后，Publish app 按钮就会被激活。

11.4.3　商店列表所需的资源

你需要上传一些图片资源（图 11-9）作为 Play 商店列表的一部分，包括：

- 至少两张应用截图；
- 一张 512 × 512 像素的 PNG 格式的应用图标；
- 一张 1024 × 500 像素的 JPG 或 PNG 格式的“功能图片”给 Play 商店。

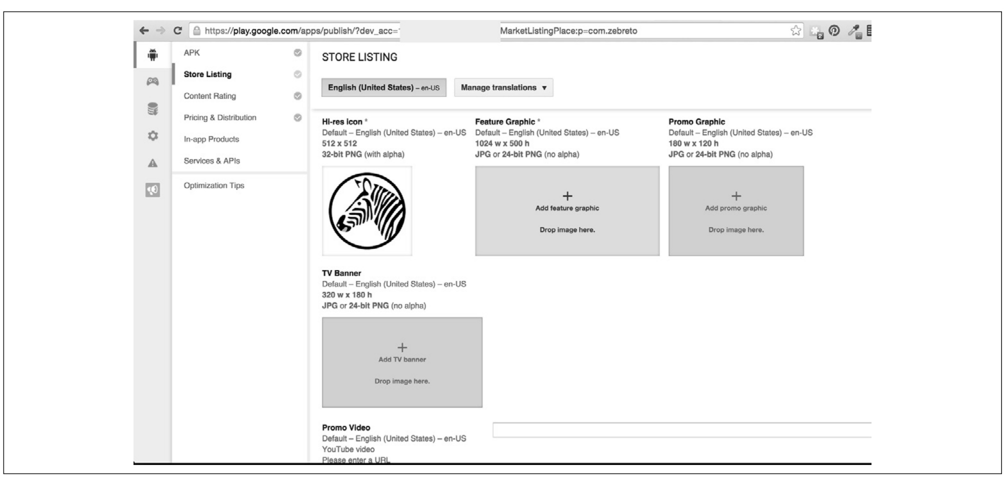


图 11-9：上传图片至 Play 商店列表

还有一些其他的可选图像，如推广图片和推广视频。

记住，这些资源非常关键，可以帮助你的应用在 Play 商店中脱颖而出！

如果你还没有屏幕截图，那么是时候去准备几张了。有两个选择：可以从物理设备获取，或者从模拟器获取截图。同时按下物理设备上的“电源键”和“音量减小键”就可以获取一张屏幕截图。

从模拟器获取屏幕截图可能有些复杂。首先，需要确保已经为模拟器分配了一个 SD 卡用来储存文件（你可以运行 `android avd`，然后点击 Edit 按钮查看模拟器的详细信息）。

接着，使用 `adb shell` 获取屏幕截图：

```
adb shell screencap -p /sdcard/screen.png
adb pull /sdcard/screen.png
adb shell rm /sdcard/screen.png
```

这些命令将会获取一张屏幕截图，然后把它保存在本地文件系统中。

如果你更喜欢单行的方式，可以试试下面的命令：[1]

```
adb shell screencap -p | perl -pe 's/\x0D\x0A/\x0A/g' > screen.png
```

这个命令将会复制你的屏幕截图到本地文件系统的 screen.png 文件中。

11.4.4　发布应用

准备好发布应用了吗？如图 11-10 所示，点击 Publish app 按钮！

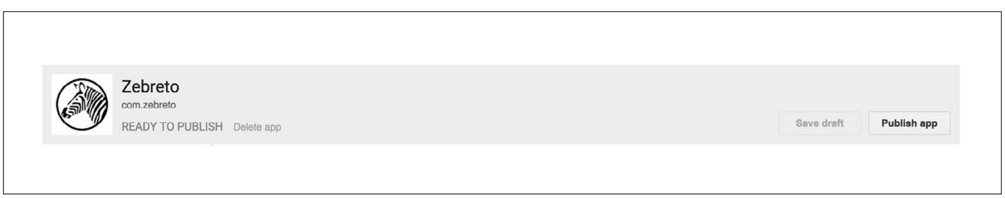


图 11-10：发布你的应用至 Play 商店

提交应用审核之后，你的应用状态将会变成“等待”（Pending），如图 11-11 所示。

注 1：单行的方式经由 shvestov’s blog 提供，如果你对我们为什么使用 Perl 感到好奇，可以打开网站看看：http://blog.shvetsov.com/2013/02/grab-android-screenshot-to-computer-via.html。——译者注

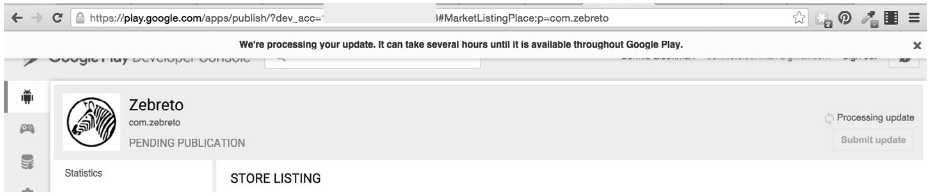


图 11-11：应用等待审核

一般来说，你会在 24 小时之内收到结果。然后，你的应用就可以通过 Play 商店提供给公众用户了，祝贺你！结果如图 11-12 所示。最后，完整版的 Android 应用已经发布成功，你可以尽情沉浸在喜悦中了。

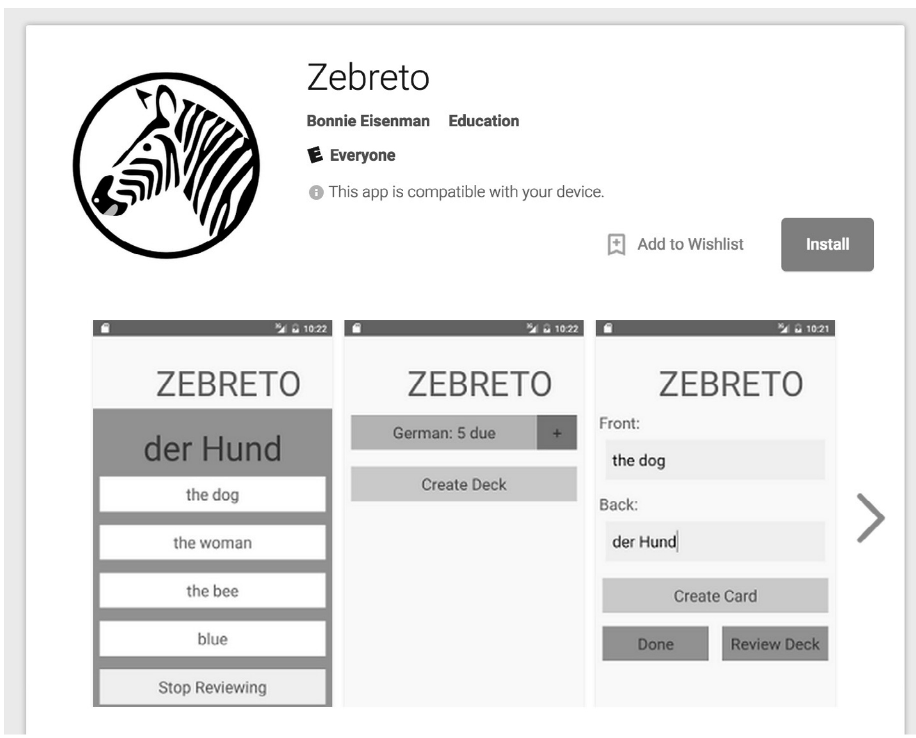


图 11-12：Zebreto 应用出现在 Play 商店中

11.5　小结

至此，你应该能够发布 React Native 应用到 Google Play 商店和 iOS 应用商店了！

总体而言，Android 应用的发布流程相比于 iOS 应用商店更为简洁。但是就像 iOS 平台一样，成功部署应用只是一个开始，你需要计划对应用后续版本的支持。并且与 iOS 平台相同的是，你不能指望用户为应用开启自动升级。此外 Android 用户有一系列的不同标准的设备，因此 Android 平台上每个版本的用户测试都至关重要。

总结

已经读到这里了，祝贺你！

我们从第一个 React Native 应用的开发开始，一路“过关斩将”，最后分别成功部署了跨平台的应用到 iOS 应用商店和 Google Play 商店。为了完成这项任务，我们从基础的 React Native 组件和样式开始学起，然后学习了触摸和平台原生接口，比如 Camera Roll（相机）和 Geolocation 地理接口。我们还掌握了通过开发者工具进行 React Native 调试的技巧以及部署应用到真实设备的方法。对于 React Native 标准库之外的功能，我们也了解了原生 Objective-C 和 Java 模块的用法，以及如何通过 npm 安装第三方 JavaScript 类库。

你所具备的 JavaScript 和 React 的知识，再加上本书中所讨论的内容，应该足以让你快速且高效地开发出适用于 Android 和 iOS 平台的跨平台应用了。当然，只拥有这些知识还远远不够，你仍需要不断学习，单单这一本书并不能涵盖 React Native 移动应用开发的方方面面。如果你遇到困难或者问题，可以向社区求助，例如 Stack Overflow（http://stackoverflow.com/questions/tagged/react-native）或 IRC（irc://chat.freenode.net/reactnative）。

让我们保持联络吧！加入 Learning React Native（http://learningreactnative.com）的邮件列表，可以获取更多关于本书的资源和更新。你也可以在 Twitter 上找到我：@brindelle（http://twitter.com/brindelle）。

最后也最重要的是，享受这一切！期待看到你们的优秀作品！

附录A

ES6语法

书中的一些代码使用了广为人知的 ES6 语法。如果你对 ES6 语法不熟悉的话也不要紧，它基本上是从你熟悉的 JavaScript 语法直接翻译过来的。

ES6 指的是 ECMAScript 6，也被称作 Harmony，是 ECMAScript 的下一代标准，JavaScript 是 ECMAScript 的具体实现。命名约定的背后有很多有趣的历史，但你只需要知道：ES6 是一个 JavaScript 的新版本，并且在继承了现有规范的前提下还引入了一些实用的新特性。

React Native 使用 Babel（https://babeljs.io/）这个 JavaScript 编译器来转换我们的 JavaScript 和 JSX 代码。Babel 的一个特点是将 ES6 语法编译成符合 ES5 规范的 JavaScript 代码，因此我们可以在 React 代码中使用 ES6 语法。

A.1　解构

解构赋值（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment）给我们提供了一个从对象提取数据的简便的方法。

这是符合 ES5 规范的代码片段：

```
var myObj = {a: 1, b: 2};
var a = myObj.a;
var b = myObj.b;
```

我们可以使用更简洁的解构赋值：

```
var {a, b} = {a: 1, b: 2};
```

你可能经常在 `require` 语句中看到它的身影。在引入 React 的时候，实际上我们取出了一个对象，并使用例 A-1 的方法来获得组件。

例 A-1：非解构语法导入 <View> 组件

```
var React = require('react-native');
var View = React.View
```

但使用解构语法会让代码更加优雅，如例 A-2 所示。

例 A-2：解构语法导入 <View> 组件

```
var { View } = require('react-native');
```

A.2 导入模块

一般情况下，我们使用 CommonJS 模块的语法导出组件以及其他一些 JavaScript 模块（例 A-3）。在该模块系统中，我们使用 `require` 语句导入其他模块，使用 `module.exports` 赋值的方法导出代码供其他模块使用。

例 A-3：使用 CommonJS 语法导入和导出模块

```
var OtherComponent = require('./other_component');

var MyComponent = React.createClass({
  ...
});

module.exports = MyComponent;
```

在 ES6 的语法（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import）中，我们采用 `export` 和 `import` 关键字来替代。例 A-4 展示了使用 ES6 模块语法编写的等效代码。

例 A-4：使用 ES6 模块语法导入和导出模块

```
import OtherComponent from './other_component';

var MyComponent = React.createClass({
  ...
});

export default MyComponent;
```

A.3 函数简写

ES6 的函数简写也是很实用的。在符合 ES5 规范的 JavaScript 中，我们用例 A-5 所示的方法来声明函数。

例 A-5：普通函数的声明方式

```
render: function() {
  return <Text>Hi</Text>;
}
```

重复使用 `function` 关键字会让人厌倦。例 A-6 展示了使用 ES6 函数简写的方式编写的相同的函数。

例 A-6：函数简写的声明方式

```
render() {
  return <Text>Hi</Text>;
}
```

A.4 箭头函数

在 ES5 版本的 JavaScript 中，为了确保函数的上下文（即 `this` 的值）符合预期，我们通常会使用 `bind` 函数（例 A-7）。这种方法在处理回调函数的时候尤为常见。

例 A-7：使用 ES5 的 JavaScript 手动绑定函数

```
var callbackFunc = function(val) {
  console.log('Do something');
}.bind(this);
```

箭头函数（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions）实现了自动绑定，因此我们不需要手动处理（例 A-8）。

例 A-8：使用箭头函数实现绑定

```
var callbackFunc = (val) => {
  console.log('Do something');
};
```

A.5 字符串插值

在 ES5 版本的 JavaScript 中，我们使用例 A-9 这样的代码来实现字符串拼接。

例 A-9：ES5 版本的 JavaScript 字符串拼接

```
var API_KEY = 'abcdefg';
var url = 'http://someapi.com/request&key=' + API_KEY;
```

ES6 为我们提供了模板字符串（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals），它支持多行字符串以及字符串插值。通过一对反引号把字符串围起来，我们就可以使用 `${}` 语法插入其他变量了（例 A-10）。

例 A-10：ES6 的字符串插值

```
var API_KEY = 'abcdefg';
var url = `http://someapi.com/request&key=${API_KEY}`;
```

附录B

命令与快速入门指南

该附录可以作为 React Native 快捷命令的参考。

B.1 创建一个新项目

```
react-native init MyProject
```

B.2 在iOS上运行

在 Xcode 中打开 ios/MyProject.xcodeproj 文件，然后在左上角点击运行按钮，可以同时启动 React Native 包管理器和 iOS 模拟器。

iOS屏幕截图

在 iOS 模拟器上按下快捷键 Command+S 就可以将屏幕截图保存到桌面上。

在物理设备中，同时按下“电源键”和“主键”也可以进行屏幕截图。

B.3 在Android上运行

首先，确保你已经连接了可识别的设备。

通过如下命令运行模拟器：

```
android avd
```

创建一个新的 Android 虚拟设备（Android Virtual Device，AVD）或选择一个存在的设备并点击 Start... 按钮。

另一种方式是在物理设备上开启 USB 调试，然后通过 USB 连接到电脑上。通过设备上的“设置 → 关于手机 → 版本号”菜单来开启 USB 调试（USB debugging）。多次重复点击“版本号”选项，直到设备询问是否开启开发者模式，然后选择“确定”。

完成上述步骤后运行：

```
react-native run-android
```

该命令将会在设备上安装你的应用，同时也会启动 React Native 包管理器。

Android屏幕截图

你可以在物理设备上同时按下“电源键”和“音量减小键”来截取屏幕。

在模拟器上截取屏幕：需要确保你的模拟器已经启用了“SD 卡存储”选项。然后运行 adb 命令：

```
adb shell screencap -p /sdcard/screen.png
adb pull /sdcard/screen.png
adb shell rm /sdcard/screen.png
```

或者使用下面这个更简洁的命令：

```
adb shell screencap -p | perl -pe 's/\x0D\x0A/\x0A/g' > screen.png
```

B.4 运行React Native包管理器

如果你由于某些原因需要手动启动 React Native 包管理器的话，可以直接进入项目的根目录，然后运行：

```
npm start
```

作者简介

Bonnie Eisenman 是 Twitter 公司的软件工程师，曾就职于 Codecademy、Google 和 Fog Creek Software 公司。她曾在多个会议上作过演讲，话题涉及 React、音乐编程和 Arduino。工作之余，她乐于开发电子乐器，喜爱使用激光切割巧克力，并且热爱学习各种语言。

关于封面

本书封面上的动物是环尾袋貂（学名：Pseudocheirus peregrinus），是澳洲土生土长的有袋类动物。环尾袋貂是食草类动物，主要居住在森林地区。它因卷曲的、末端经常呈环状的尾巴而得名。

环尾袋貂外表是灰棕色的，体长可达 35 厘米。它以各式各样的叶子、花朵和水果为食。环尾袋貂是夜行性动物，并群居在巢穴中。作为有袋类动物，环尾袋貂会把它们的幼崽置于袋中，直到幼崽长大，可以独立生存。

20 世纪 50 年代，环尾袋貂数量骤减，好在近年来数量又有所增加。但由于森林砍伐，它们的栖息地仍然受到威胁。

O’Reilly 书籍封面上的许多动物都属于濒危物种，它们都是这个世界不可或缺的一部分。想知道如何帮助这些动物的话，请访问 http://animals.oreilly.com。

本书封面图片来自 *Shaw’s Zoology*。

延 展 阅 读



国内第一本 React Native 原创图书，携程框架团队倾力打造
帮助零基础读者实现跨终端 App 开发

书号 : 978-7-115-41191-4
定价 : 79.00 元



使用各种 Java 工具、客户端技术和 Web API 来开发 Web 应用

书号 : 978-7-115-39730-0
定价 : 59.00 元



前段开发人员进阶首选
掌握 JavaScript 语言各种细节，轻松编写出易维护、易扩展的高质量代码

书号 : 978-7-115-40255-4
定价 : 79.00 元



Amazon 榜首畅销书全新升级，Android 入门进阶不二之选
全面覆盖 Android 开发知识点，通过实战项目手把手教你逐步写 Android 应用

书号：978-7-115-42246-0
定价：109.00 元

O'REILLY®

React Native开发指南

自2015年春天Facebook开源React Native以来，React Native就凭借其强大的可扩展性、良好的用户体验以及可拥有原生外观等优势得到开发者的关注和青睐。

本书是一本实践指南，从基础知识入手，逐步深入，带领读者部署可100%代码复用的、成熟的跨平台移动应用。作者通过示例代码向Web开发者和前端工程师展示了如何使用移动组件构建界面并编写样式，以及如何调试和部署应用。除了框架本身的讲解，作者还探讨了如何使用第三方库，以及如何编写自己的Java或者Objective-C的React Native扩展。

- 了解React Native如何开放原生UI组件接口
- 类比HTML元素，了解该框架如何使用原生组件
- 创建自己的React Native组件和应用，并为它们编写样式
- 为该框架不支持的API和功能安装第三方模块
- 使用工具来调试代码，并解决JavaScript之外的问题
- 整合所学知识，开发一款高效记忆闪卡应用——Zebreto
- 部署应用至iOS应用商店和Google Play商店

Bonnie Eisenman，Twitter公司软件工程师，曾就职于Codecademy、Google和Fog Creek Software公司。她曾在多个会议上作过演讲，话题涉及React、音乐编程和Arduino。

“Bonnie在书中讲解了React Native最重要的知识，帮助你建立扎实的开发基础！认真研读本书后，你将拥有足够的信心继续去探索并学习更多的知识。”

——Brent Vatne
Exponent开发者，
React Native核心贡献者

JAVASCRIPT

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／软件开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-42526-3

ISBN 978-7-115-42526-3
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks